Robotski manipulator za ispisivanje fotografija

Paradžik, Ivan

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Ivan Paradžik

Zagreb, godina 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Tomislav Stipančić

Student:

Ivan Paradžik

Zagreb, godina 2020.

Izjavljujem da sam ovaj rad izradio samostalno koristeći stečena znanja tijekom studija i navedenu literaturu.

Zahvaljujem se Doc. dr. sc. Tomislavu Stipančiću na pruženoj pomoći i korisnim savjetima. Također se zahvaljujem obitelji i prijateljima na pruženoj podršci tijekom cijelog školovanja.

Ivan Paradžik

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske radove studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala te mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum:	Prilog:
Klasa: 602 - 04 / 20 - 6 / 3	
Ur. broj: 15 - 1703 - 20 -	

DIPLOMSKI ZADATAK

Student: **IVAN PARADŽIK** Mat. br.: 0035196797

Naslov rada na hrvatskom jeziku: **Robotski manipulator za ispisivanje fotografija**

Naslov rada na engleskom jeziku: **A robotic manipulator for printing photos**

Opis zadatka:

U radu je potrebno osmisliti i napraviti crtač - robotski manipulator sastavljen od elektroničkih komponenti (mikroprocesor i električni motori) te jednostavne konstrukcije. U sklopu softverskog rješenja potrebno je:
1. Osmisliti i implementirati matematički model koji definira položaj pisača (kemijske olovke) u prostoru u odnosu na prvi motor koji predstavlja ishodište koordinatnog sustava.
2. Točke gibanja definirati fotografijom koju crtač treba nacrtati na papirnatoj površini. Koristeći razvijenu računalnu podršku, fotografije je najprije potrebno grafički obraditi (pojednostavniti) te prilagoditi za crtanje.
3. Implementirati odgovarajući mrežni protokol za slanje podataka na mikrokontroler koji će upravljati motorima crtača.
4. Softversku podršku temeljiti na Python programskom okruženju te pripadajućim programskim bibliotekama.
Razvijeno cjelovito rješenje potrebno je eksperimentalno evaluirati. U radu je potrebno navesti korištenu literaturu te eventualno dobivenu pomoć.

Zadatak zadan:
5. ožujka 2020.

Rok predaje rada:
7. svibnja 2020.

Predviđeni datum obrane:
11. svibnja do 15. svibnja 2020.

Zadatak zadao:

doc. dr. sc. Tomislav Stipančić

Predsjednica Povjerenstva:

prof. dr. sc. Biserka Runje

SADRŽAJ

POPIS SLIKA

POPIS OZNAKA

Oznaka	Jedinica	Opis
$\overline{AB}$	cm	Dužina unutarnje ruke manipulatora
$\overline{BC}$	cm	Dužina vanjske ruke manipulatora
$\overline{AC}$	cm	Najbliža udaljenost kemijske olovke od ishodišta manipulatora
trenutni_x	cm	Položaj kemijske olovke u osi x koordinatnog sustava robota
trenutni_y	cm	Položaj kemijske olovke u osi y koordinatnog sustava robota
α_0	°	Početni kut prvog koračnog motora
β_0	°	Početni kut drugog koračnog motora
γ_0	°	Početni kut trokutnog mehanizma u točki A
ε_0	°	Početni kut trokutnog mehanizma u točki B
δ_0	°	Početni kut trokutnog mehanizma u točki C
α	°	Kut prvog koračnog motora
β	°	Kut drugog koračnog motora
γ	°	Kut trokutnog mehanizma u točki A
ε	°	Kut trokutnog mehanizma u točki B
δ	°	Kut trokutnog mehanizma u točki C
Ω	°	Kut koji dužina $\overline{AC}$ zatvara s pozitivnom osi x

POPIS TABLICA

SAŽETAK

Kroz ovaj rad je osmišljen i napravljen niskobudžetni manipulator s tri stupnja slobode gibanja koji služi za ispisivanje fotografija. Manipulator se sastoji od jednostavne konstrukcije povezane sa sklopovskom podrškom koja omogućuje pokretanje mehanizma manipulatora. Sklopovska podrška sastoji se od dva koračna motora s upravljačkim sklopovima koji služe za rotaciju ruku manipulatora te servo motora koji služi za podizanje i spuštanje kemijske olovke od podloge manipulatora. Mikrokontroler koji se koristi za upravljanje manipulatorom je *Raspberry Pi Zero W*. Napravljena je programska podrška upravljanja manipulatora temeljena na izvedenom matematičkom modelu kojim se definira položaj kemijske olovke u koordinatnom sustavu manipulatora. Programska podrška manipulatora pisana je u *Python* programskom jeziku. Napravljena je i programska podrška obrade fotografije. Rezultati obrade fotografije koriste se za upravljanje manipulatorom na načina da se manipulatoru redom zadaju točke po kojima se mora gibati kako bi ispisao fotografiju. U konačnici su prikazani eksperimentalni rezultati obrade fotografije i ispisa obrađene fotografije pomoću napravljenog robotskog manipulatora.

Ključne riječi: manipulator, koračni motor, servo motor, *Raspberry Pi Zero W, Python ,* obrada fotografije

SUMMARY

The goal of this master's thesis is to design and construct low cost robotic manipulator with three degrees of freedom whose purpose is to print photos. Simple construction is connected with hardware system that enables control of manipulator mechanism. Hardware parts used for this project are two stepper motors with motor drivers which enable rotational movement of the manipulator arms, and one servo motor that enables lowering and lifting of the pen from the drawing surface. Microcontroller used for controlling the manipulator is *Raspberry Pi Zero W.* Implemented control software uses mathematical model which defines the position of the pen in coordinate system of manipulator. Control software uses result of the image processing software, which represents the path that the manipulator must traverse in order to print a photo. Both softwares are implemented in Python programming language. Finally, experimental results of image processing as well as images, printed by the manipulator using those results, are shown.

Key words: Manipulator, Stepper Motor, Servo Motor, Raspberry Pi Zero W, Python, Image Processing

1. UVOD

U raznim područjima industrije sve se više koriste robotski manipulatori. Jedan od popularnijih manipulatora danas su razni 3D printeri koji su cijenom postali sve više dostupni te se osim u ozbiljnim granama industrije mogu koristiti u zabavne i umjetničke svrhe. Cilj ovog diplomskog rada je napraviti robotski manipulator koji bi služio za ispisivanje fotografija. Iako bi se takav manipulator lako mogao napraviti pomoću malih modifikacija na manipulatoru kao što je 3D printer, može se reći da je cijena 3D printera i dalje previsoka za takav projekt. Stoga je potrebno napraviti manipulator za ispisivanje fotografija koji će cijenom biti puno jeftiniji od 3D printera. Cijena samih elektroničkih komponenata koje se koriste za izradu manipulatora iznose manje od 10 dolara za sve komponente zajedno te se one prvenstveno koriste za učenje programiranja mikrokontrolera.

Izuzmemo li programiranje i elektroniku, za izradu ovog diplomskog rada iskorištene su i ostale grane strojarstva kao što su konstruiranje, matematika, mehanika te računalna obrada slike. Tako je za početak potrebno napraviti konstrukciju manipulatora od jednostavnih i pristupačnih dijelova koji se nalaze oko nas u svakodnevnom životu. Samu konstrukciju je potrebno povezati s motorima koji imaju ulogu pokretanja mehanizma manipulatora. Nakon toga treba spojiti elektroničke elemente, odnosno dva koračna motora i jedan servomotor na mikrokontroler. Mikrokontroler koji se koristi je danas najpopularniji mikrokontroler *Raspberry Pi* koji je zapravo malo računalo koje omogućuje pristup pinovima za programiranje elektroničkih komponenata [1]. Budući da je cilj napraviti niskobudžetni manipulator, korišten je model *Raspberry Pi Zero W* koji je cijenom najpristupačniji na tržištu. Osnovnom matematikom je potrebno raspisati matematički model koji opisuje položaj kemijske olovke, koja služi za ispisivanje fotografija, u odnosu na koračni motor koji predstavlja ishodište koordinatnog sustava manipulatora. Sljedeći korak je izrada programske podrške manipulatora u programskom jeziku *Python* kojom se može kontrolirati motore manipulatora. U programsku podršku potrebno je implementirati matematički model kako bi se omogućilo pokretanje manipulatora u x-y koordinatnom sustavu manipulatora. Također je potrebno napraviti programske podršku obrade fotografije. Fotografije se obrađuju tako da se koristiti slikarska tehnika sjenčanja (*eng.*

hatching) kojom se pomoću crtanja paralelnih linija dobiva određeni ton boje, odnosno sjene koja ovisi o gustoći linija. Ovaj manipulator ima mogućnost ispisivanja dva tona boje, ovisno o boji kemijske olovke koja se koristi. Rezultati obrade fotografije koriste se kao ulaz u programsku podršku upravljanja manipulatora te se dolazi do konačnog cilja rada, ispisivanja fotografija.

2. RAČUNALNA SKLOPOVSKA PODRŠKA MANIPULATORA

U ovom poglavlju su opisane pojedine elektroničke komponente za izvedbu manipulatora. Elektroničke komponente koje su potrebne za izvedbu crtača su dva koračna motora *28BYJ-48* za koja su potrebna dva upravljačka sklopa za motore *ULN2003A*, jedan mikro servo motor *Tower Pro SG90* te najvažnija komponenta mikrokontroler, odnosno malo računalo koje ima mogućnost programiranja raznih komponenata preko pinova, *Raspberry Pi Zero W*. Osim toga koriste se i obične žice za spajanje komponenata s *Raspberry Pi-om.*

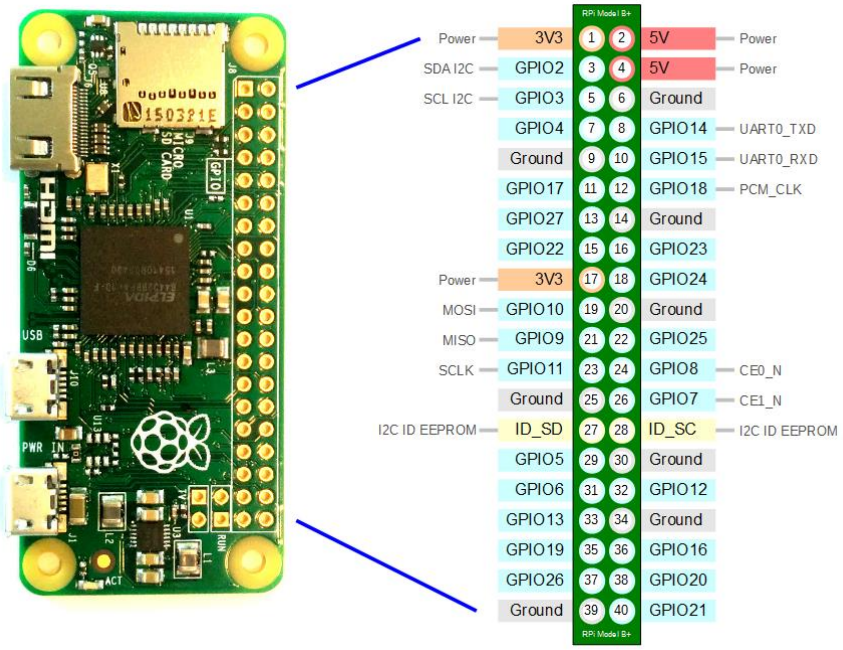
2.1. *Raspberry Pi Zero W*

Raspberry Pi je malo računalo sastavljano od jedne ploče na kojoj se nalaze sve komponente. Razvijeno je u Velikoj Britaniji s ciljem podupiranja učenje osnova računalnih znanosti, ali i mehatronike, robotike i strojarstva [1]. Budući da je cilj rada napraviti niskobudžetni crtač, koristi se jedan od jeftinijih *Raspberry Pi*-ova dostupnih na tržištu: *Raspberry Pi Zero W*(slika 2.1.). Cijena ovog *Raspberry Pi*-a na tržištu je tek deset dolara, a može se nabaviti i za manje od deset dolara[2].

Slika 2.1. *Raspberry Pi Zero W* [2]

Raspberry Pi Zero W je i najmanji mogući *Raspberry Pi* dimenzija od 60x30x5 milimetara. Zbog svoje veličine ima smanjen broj ulaza i izlaza za interakciju s drugim računalima te za razliku od standardnog *Raspberry Pi* nema *USB i Ethernet* ulaze. Također postoje i dva

micro USB ulaza od kojih je jedan za napajanje *Raspberry Pi-a*, dok drugi služi za komunikaciju s drugim računalom preko serijskog porta, *mini HDMI* ulaz pomoću kojeg je moguće *Raspberry Pi* spojiti s ekranom koji omogućuje direktan pristup *Raspberry Pi* sustavu te ulaz za kameru. Procesor na *Raspberry Pi Zero W-u* razvile su organizacije *Broadcom Inc.* zajedno s *Raspberry Pi Foundation*. Sustav na čipu je isti kao kod prve generacije *Raspberry Pi-a*, a to je *Broadcom BCM2835 SoC* (*System On Chip*) s procesorom *ARM11* koji radi na radnom taktu od 1GHz. *Raspberry Pi Zero W* ima radnu memoriju (*RAM*) od 512 *MB*. Jedna od prednosti ovog *Raspberry Pi* je ta što ima mogućnost bežičnog spajanja na internet te *Bluetooth* [3]. Na Slici 2.2. vidimo prikaz pinova za ulaz i izlaz za generalnu upotrebu preko kojih je moguće spajati elektroničke komponente i njima upravljati preko *Raspberry Pi*-a.



Slika 2.2. *GPIO* pinovi, *Raspberry Pi Zero W*

Iako ima smanjen broj ulaza i izlaza, i dalje posjeduje ulaze i izlaze za generalnu upotrebu *(eng. General-purpose input/output)* koji su najbitniji za ovaj projekt jer omogućuju komunikaciju s raznim elektroničkim komponentama kao što su koračni motori i servo motori. *Raspberry Pi Zero W* posjeduje 40 pinova za ulaze i izlaze [4]. Od tih 40 pinova, 25 pinova je za generalnu upotrebu *(GPIO)*. Također postoje dva pina koji omogućuju napajanje komponenata sa strujom od 5 volta te dva pina koja služe za napajanje komponentata od 3.3 volta[4]. Naravno, postoji i pet pinova koji služe za uzemljenje tih istih komponenata. Ostali pinovi služe za razna alternativna upravljanja

elektroničkih komponenata[4]. U kasnijim poglavljima će biti objašnjen način rada i način upravljanja *GPIO* pinova.

Raspberry Pi Zero W također ima utor za *microSD* karticu na koju je potrebno postaviti operativni sustav. Najčešći operativni sustav koji se postavlja na *Raspberry Pi* zove se R*aspbian*. *Raspbian* je operativni sustav temeljen na *Debianu* pa se sam sustav može usporediti s *Linuxom* [3]. Za ovaj projekt je na *Raspberry Pi Zero W* instaliran operativni sustav *Raspian Buste Lite* kojeg je potrebno postaviti na *microSD* karticu [3]. Kako *Raspian Buste Lite* nema mogućnost direktnog pristupa sučelju u nastavku će biti objašnjeno kako pristupiti *Raspberry Pi-u* preko drugog računala preko *SSH(Secure Shell)* mrežnog protokola.

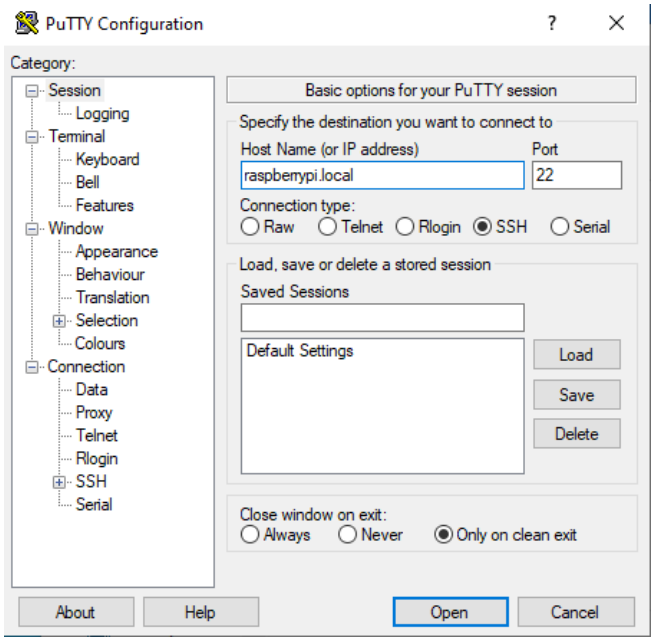
2.1.1. *SSH (Secure Shell)* mrežni protokol

SSH je mrežni protokol koji korisnicima omogućuje uspostavu sigurnosnog komunikacijskog kanala između dva računala [5]. *SSH* se temelji na korištenju metoda asimetrične i simetrične kriptografije koje omogućuju sigurni prijenos podataka računalnom mrežom. Sami *SSH* obično korist porti 22 što je u startu postavljeno u svim aplikacijama. Kako bi se pristupilo *Raspberry Pi-u* preko *SSH* mrežnog protokola prije svega je potrebno na *microSD* karticu postaviti datoteku koju je potrebno nazvati ssh te datoteka ne smije imati nikakav datotečni nastavak [5]. Budući da je cilj imati bežičnu komunikaciju između *Raspberry Pi-a* i računala također je potrebno na *microSD* karticu postaviti datoteku koja će u sebi posjedovati naredbu koja *Raspberry Pi* spaja automatski na bežičnu mrežu na koju će i računalo biti spojeno. Datoteka treba biti nazvana *wpa_supplicant.conf* te se u njoj treba nalaziti:

```
country=CRO
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
network={ssid="Ime_interneta"
        psk="Lozinka_interneta"}
```

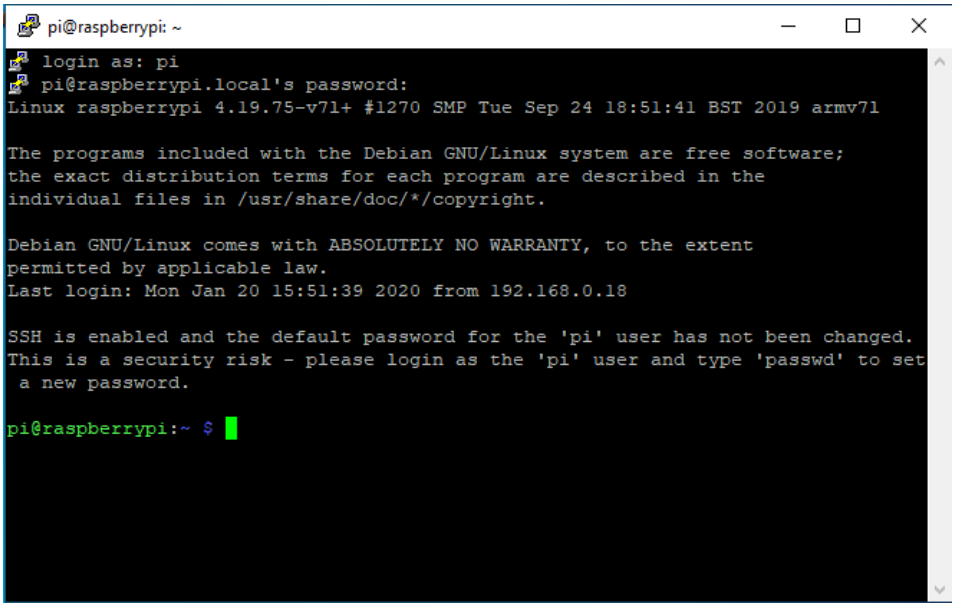
Na računalo preko kojeg se želimo spojiti je potrebno instalirati aplikaciju preko koje se uspostavlja veza između dva računala. Postoje razne aplikacije za *SSH* mrežni protokol.

U ovom radu se koristi najpopularnija besplatna aplikacija koja se koristi za komunikaciju s *Raspberry Pi-o* , a to je *PuTTY* [5]. Osim *SSH* mrežnog protokola ova aplikacije omogućuje i ostale načine spajanja *Raspberry Pi* na drugo računalo kao na primjer preko serijskog porta micro *USB*-a. Na slici 2.3. prikazan je prozor nakon otvaranja same aplikacije.



Slika 2.3. Aplikacija *PuTTY* za *SSH* mrežni protokol

Na slici se vidi da je za uspostavu komunikacije potrebno upisati *IP* adresu lokalne mreže na koju je spojen *Raspberry Pi* ili ime uređaja na koji se želimo spojiti koji je u ovom slučaju *raspberrypi.local*. Port je, kao što je već rečeno, postavljen na 22. Nakon unosa imena računala ili *IP* adrese lokalne mreže može se pristupiti *Raspberry Pi-u*, odnosno prozoru za upravljanje *Raspberry Pi-om* koji je prikazan na slici 2.4.. U ovom prozoru se u konačnici omogućuje upravljanje samim robotskim manipulatorom.



Slika 2.4. Prozor za upravljanje *Raspberry Pi-om*

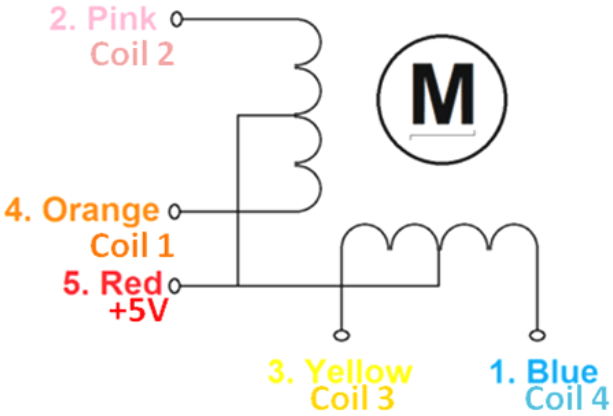
2.2. Unipolarni koračni motori *28BYJ-48*

Budući da su glavni dijelovi samog robotskog manipulatora koračni motori potrebno je opisati što su to koračni motori kako bi kasnije bilo jasnije kako upravljati njima. Koračni motori su zapravo elektromehanički pretvornici energije [7]. Njihova je uloga da električnu pobudu pretvaraju u mehanički pomak koji je definiran koracima. Pomak može biti rotacijski ili translacijski ovisno o samoj mehanici. Koračni motori su zapravo obični istosmjerni motori bez četkica koji se, za razliku od istosmjernog motora s četkicama, ne rotiraju kontinuirano nego se rotiraju preko koraka kojima je moguće upravljati i zadržati ih u istoj poziciji bez ikakvog senzora pozicije i povratne veze [7]. Naravno koračni motor se mora pravilno dimenzionirati s okretnim momentom i brzinom kako bi ti koraci bili pravilni. Postoje razne vrste koračnih motora, ali glavna podjela je na bipolarne i unipolarne koračne motore. Kako je cilj diplomskog rada napraviti niskobudžetni crtač, za rad se koristite dva unipolarna koračna motora *28BYJ-48* (slika 2.5.) koje je moguće nabaviti za samo dolar te se oni pretežno koriste za učenje programiranja elektroničkih komponenata. Na slici možemo vidjeti da sami motor ima 5 žica što je glavni znak da se radi o unipolarnom motoru pa je stoga za upravljanje potreban upravljački sklop za unipolarne motore koji je dosta jeftiniji i jednostavniji od upravljačkih sklopova za upravljanje bipolarnim motorima. Upravljački

sklop koji se koristit za upravljanje posjeduje red tranzistora *ULN2003A* koji je u nastavku opisan.

Slika 2.5. Unipolarni koračni motor *28BYJ-48* [8]

Koračni motor rotira pokretni dio preciznim koracima u određenim vremenskim intervalima. Puls se šalje do serije zavojnica koje čine prsten oko rotora, ali one same su statične pa taj dio motora zovemo stator [8]. Ovaj unipolarni motor ima četiri zavojnice organizirane u grupe koje predstavljaju faze (slika 2.6.). Faze zapravo predstavljaju smjer magnetskog polja i okidanjem tih zavojnica može se mijenjati smjer gibanja motora [8]. Sami način slanja signala zavojnicama, odnosno upravljanja koračnim motorima, bit će objašnjen u idućim poglavljima.



Slika 2.6. Shematski prikaz unipolarnog koračnog motora *28BYJ-48* [8]

Preciznost motora se izražava brojem koraka koje motor napravi kako bi učinio puni krug, odnosno 360°. Sama preciznost se naziva rezolucija koračnog motora. Pomoću

podataka o koračnom motoru može se izračunati broj koraka koji je potreban da se motor okrene za svih 360°. Potrebni podatci za izračun broja koraka su omjer okretanja, odnosno prijenosni omjer zupčanika, te stupanj jenog okreta. Prijenosni omjer ovog motora je 1/16, dok je stupanj jednog okreta 11.25°. Stoga se pomoću sljedeće formule može izračunati broj koraka koračnog motora:

Broj korak = (360°/stupanj jednog okreta)*prijenosni omjer

Broj koraka = 512

U konačnici možemo dobiti broj stupanjeva za koji se motor pomakne kada napravi jedan korak na način da 360° podijelimo sa brojem koraka. Dobiva se da za jedan korak motor pomakne 0.703125° što je ključan podatak za programiranje robotskog manipulatora.

Ostale bitnije karakteristike motora su[8]:

- Napon : 5 VDC
- Frekvencija rada : 100Hz
- DC otpor : 50 Ω
- Dinamički moment okreta: 300 gf/cm

2.2.1. Upravljački sklop za unipolarne koračne motore *ULN2003A*

ULN2003A je zapravo niz *Darlington* tranzistora koji može raditi na visokoj voltaži od 50 V i struji od 500mA [9]. *ULN2003A* sadrži 7 *NPN Darlington* tranzistora koji se mogu upravljati neovisno jedan o drugome te se najčešće koristi za upravljanje unipolarnih koračnim motorima, a može služiti i za upravljanje nekih drugih jednostavnih elektroničkih komponenata [9]. Moguće je koristiti sedam tranzistora istovremeno. Za unipolarni koračni motor koji se koristi je potrebno četiri tranzistora jer unipolarni koračni motor *28BYJ-48* posjeduje četiri faze koje je potrebno okidati u određenom sekvencijskom slijedu kako bi motor napravili jedan korak. Na slici 2.6. je prikazan upravljački sklop za unipolarne motore koji na sebi ima niz tranzistora *ULN2003A* te je već prilagođen za spajanje unipolarnog motora. Prilagođeni upravljački sklop ima poseban izlaz za unipolarne koračne motore *28BYJ-48* na koji se motor jednostavno može spojit. Postoje i četiri ulaza IN1, IN2, IN3 i IN4 koji se povezuju s

Raspberry Pi te se pomoću njih upravlja samim motorom. Naravno, postoje i ulazi za napajanje motora te za uzemljenje. Detaljni opis spajanja koračnog motora s upravljačkim sklopom i način upravljanja je objašnjen u idućim poglavljima.

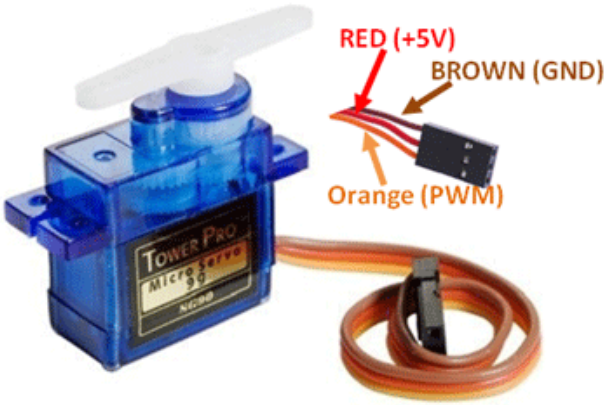
Slika 2.7. Upravljački sklop za unipolarne koračne motore *ULN2003A* [9]

2.3. Servo motor *Tower Pro SG90*

Servo motori su elektromehanički pretvornici za preciznu kontrolu kutne pozicije, brzine i ubrzanja [9]. Sastoje se od samog pretvornika i senzora koji u povratnoj petlji daje informacije o poziciji motora. Ukoliko se kut vratila motora ne nalazi na željenoj ili referentnoj poziciji, regulator povećava snagu na ulazu i vraća vratilo u tu poziciju [9]. Servo motori su zbog svoje složenosti skupi, ali nude mnoge prednosti u odnosu na druge motore. Kod servo motora za razliku od koračnih motora ne postoji opasnost od pogreške u pozicioniranju zbog same povratne veze koju motor posjeduje. Kako su servo motori dosta skupi, a uloga samog servo motora u radu nije zahtjevna, za izradu manipulatora se koristi najjeftiniji mogući mikro servo motor *Tower Pro SG90* (slika 2.8.) koji se može kupiti za manje od dva dolara. Za razliku od koračnih motora, ovaj servo motor već ima ugrađen upravljački sklop koji omogućuje upravljanje motorom. Sami motor ima tri izlaza. Crvena žica je za napajanje, smeđa žica je za uzemljenje, dok je narančasta za kontrolu samog motora [10]. Ovaj servo motor se

kontrolira pomoću pulsno širinske modulacije *(eng. Pulse Width Modulation)* signala koja je objašnjena u kasnijim poglavljima. Motor se može okretati za 180°. Ostale karakteristike ovog servo motora su [10]:

- Kontrola: analogna
- Napon: 5VDC
- Brzina: 60 stupnjeva za 0.1s
- Težina: 9 g
- Dimenzije: 23 x 12 x 29 mm



Slika 2.8. Servo motor *Tower Pro SG90* [10]

3. KONSTRUKCIJA ROBOTSKOG MANIPULATORA

U ovom dijelu rada je opisano koji su konstrukcijski dijelovi potrebni za izvedbu manipulatora. Također je objašnjen način pravljenja same konstrukcija robotskog manipulatora za ispisivanje fotografije. U konačnici se objašnjava i kako ožičiti samu konstrukciju s mikrokontrolerom.

3.1. Konstrukcijski dijelovi manipulatora

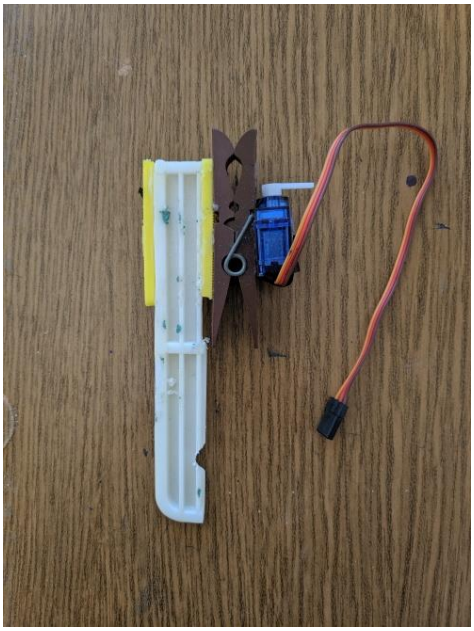
Konstrukcijski dijelovi manipulatora su dijelovi koje je moguće nabaviti u svakodnevnom životu budući da se radi o jako malim silama te zbog toga nije potrebno koristiti posebne materijale za konstruiranje. Kako je cilj konstruirati robotski manipulator za ispisivanje fotografija, prije svega je potrebna nekakva podloga na koju se postavlja konstrukcija manipulatora. Podloga mora biti ravna te bez ikakvih oštećenja kako bi kemijska olovka mogla nesmetano pisati po papiru koji je postavljen na podlogu. Dimenzija same podloge bi trebala biti minimalnih dimenzija od 30x30 centimetara kako bi na nju stala konstrukcija, elektroničke komponente i papir po kojem se ispisiva fotografija, sama debljina nije važna. Na slici 3.1. je prikazana podloga robotskog manipulatora koja se koristiti u ovom radu.

Slika 3.1. Podloga robotskog manipulatora

Osim podloge crtača potrebna su i dva štapa koji opisuju mehanizam manipulatora te predstavljaju dvije ruke manipulatora. Štapovi isto kao i sama podloga mogu biti od bilo kojeg materijala. Jedini zahtjev je taj da je štap napravljen od krutog materijala kako nebi bio podložan savijanju pri malim silama. U ovom radu se koriste štapovi od plastike. Dužina štapa je oko deset centimetara, širina oko 1.5 centimetara , a debljina do pola centimetra. Na slici 3.2. prikazan je štap unutarnje ruke manipulatora. Štap unutarnje ruke manipulatora mora na sebi imati zalijepljena dva uglavljenja za koračne motore budući da se na njega spajaju dva koračna motora. Uglavljena su također napravljena od plastike. To su ustvari uglavljenja za mikro servo motor *Tower Pro SG90* koja se mogu spojiti i na vratilo rotora koračnog motora. Uglavljenja su postavljena na udaljenost od oko 9 centimetara jedan od drugog.

Slika 3.2, Štap unutarnje ruke s uglavljenjima za motore

Na slici 3.3. imamo prikazan drugi štap, odnosno štap koji predstavlja vanjsku ruku manipulatora. Na štap je potrebno zalijepiti plastičnu pločicu koja se, kao što se vidi na slici, nalazi bočno na samom kraju štapa. Plastična pločica zapravo predstavlja nosač. Na mjestu gdje je nosač postavlja se obična kvačica(štipalica) koja služi kao držač kemijske olovke. Samo spajanje izvedeno je ljepilom. Na slici se još vidi da je na kvačicu zalijepljen i mikro servo motor s plastičnim dodatkom na vratilu rotora motora koji služi za podizanje i spuštanje kemijske olovke na podlogu manipulatora.



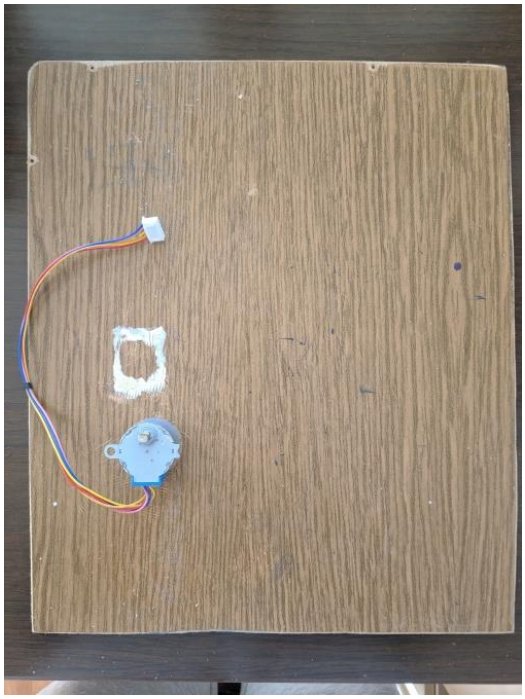
Slika 3.3. Vanjska ruka manipulatora s držačem za kemijsku olovku i servo motorom

Za izvedbu konstrukcije je naravno potrebna i obična kemijska olovka koja se postavlja u kvačicu kako se tijekom rada manipulatora kemijska ne bi pomicala i kako bi bilo moguće mijenjati kemijske olovke nakon što se potroše. Također, koristi se i dodatna mala ploča za ožičavanje elektroničkih komponenata te same žice s kojima se povezuju elektronske komponente na mikrokontroler.

3.2. Sastavljanje robotskog manipulatora

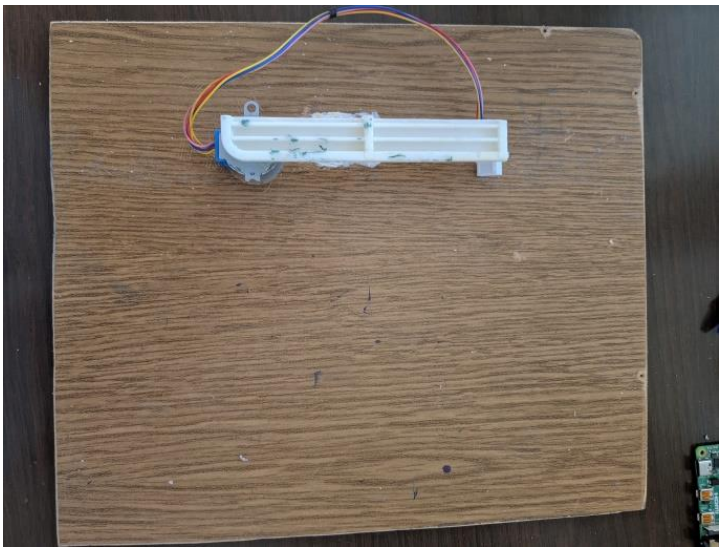
U ovom dijelu je objašnjeno kako se elektronički i konstrukcijski dijelovi spajaju u cjelinu kako bi došli do konačne konstrukcije robotskog manipulatora za ispisivanje fotografija.

Kako je već rečeno, cijelu robotsku konstrukciju je potrebno postaviti na drvenu ploču. Prvo se na ploču postavlja koračni motor koji predstavlja ishodište koordinatnog sustava manipulatora (slika 3.4.). Veza između motora i ploče se postiže lijepljenjem snažnim adhezivom. Između motora i ploče je potrebno postaviti malu pločicu koja za koji milimetar podiže motor od površine glavne ploče. Podizanjem prvog motora je potrebno jer zapravo utječe na podizanje drugog motora koji je pomičan te se s tim smanjuje trenje drugog motora s podlogom manipulatora.

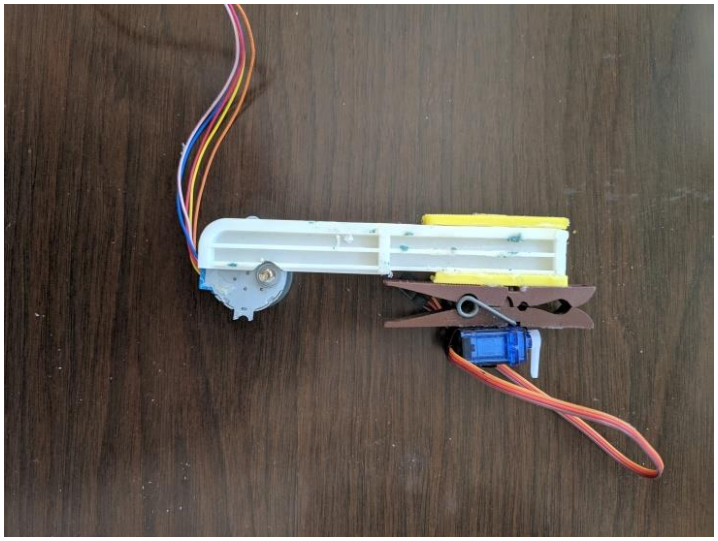


Slika 3.4. Spoj prvog koračnog motora s podlogom manipulatora

Nakon postavljanja koračnog motora, koji predstavljanje ishodište koordinatnog sustava manipulatora, potrebno je na drvenu ploču povezati štap unutarnje ruke manipulatora s prvim koračnim motorom (slika 3.5). Veza između vratila rotora koračnog motora i štapa se postiže tako što se vratilo uglavljuje na plastični ležaj, odnosno na jedno od uglavljenja postavljeno na unutarnju ruku manipulatora. Dodatna sigurnost između uglavljenja na štapu i rotora koračnog motora postiže se lijepljenjem budući da samo uglavljenje nije za koračne motore nego je namijenjeno za mikro servo motor *Tower Pro SG90*. S dobivenom vezom unutarnje ruke manipulatora i koračnog motora dobivaju se prve konture manipulatora. Rotacijom samog motora postiže se okretanje unutarnje ruke manipulatora oko ishodišta koordinatnog sustava pa se može reći da se dobiva prvi stupanj slobode gibanja manipulatora. Kako bi dobili dodatne stupnjeve slobode gibanja manipulatora, potrebno je povezati vanjsku ruku manipulatora s unutarnjom rukom manipulatora. Vanjska ruka manipulatora se prije svega spaja s drugim koračnim motorom (slika 3.6.). Kraj štapa vanjske ruke koji je slobodan te se na njemu ne nalazi nosač za kemijsku olovku, lijepi se na ravnu površinu koračnog motora gdje se nalazi vratilo rotora te se tako dobiva veza između drugog koračnog motora i vanjske ruke.



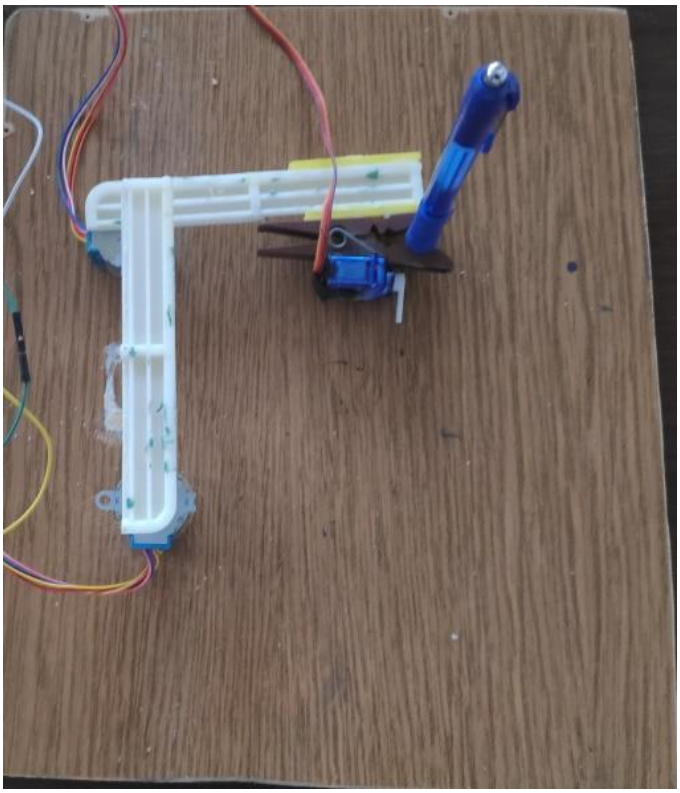
Slika 3.5. Spoj koračnog motora sa štapom unutarnjem ruke manipulatora



Slika 3.6. Spoj štapa vanjske ruke manipulatora s drugim koračnim motorom

U konačnici je potrebno spojiti unutarnju ruku manipulatora s rotorom drugog koračnog motora koji je povezan sa štapom vanjske ruke manipulatora (slika 3.7.) . Veza između koračnog motora i unutarnje ruke se postiže isto kao s prvim koračnim motorom. Na kraju ruke se nalazi uglavljenje u koje treba postaviti vratilo rotora drugog koračnog motora te se lijepljenjem uglavljenja s rotorom koračnog motora postiže sigurna veza. Spajanjem vanjske ruke manipulatora s unutarnjom rukom manipulatora dobiva se konačna konstrukcija manipulatora. Drugi koračni motor sada ima mogućnost okretanja vanjske ruke oko svoje osi s čime se postiže još jedan stupanj slobode gibanja manipulatora. Treći stupanj slobode gibanja manipulatora postiže se sa servo motorom

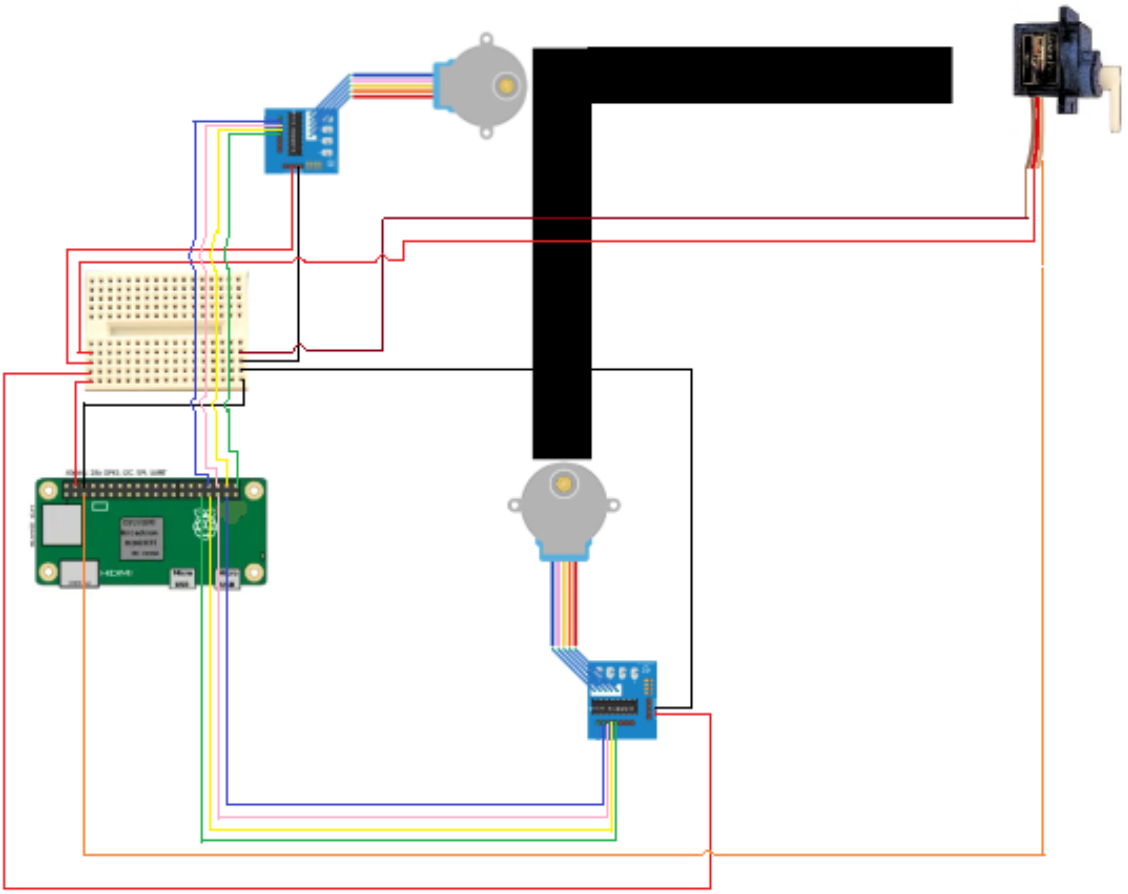
na kraju štapa vanjske ruke manipulatora. Servo motor rotacijom plastičnog nastavka na rotoru motora može spuštati kemijsku olovku na podlogu, odnosno podizati kemijsku olovku od podloge manipulatora. Na kraju, možemo zaključiti da ovakvom konstrukcijom manipulatora uspješno dobivamo tri stupnja slobode gibanja manipulatora. Dva stupnja slobode koje omogućava koračni motori su rotacijski, dok je stupanj slobode koji omogućava servo motor translacijski.



Slika 3.7. Konačna konstrukcija manipulatora

3.3. Ožičavanje robotskog manipulatora

Kako bi se napravila konačna konstrukcija manipulatora, preostalo je spojiti elektroničke komponente s *Raspberry Pi-om*. Na Slici 3.8. prikazana je shema spajanja svih komponenata s mikrokontrolerom. Za bolje razumijevanje kako se točno spajaju pojedine elektroničke komponente na *Raspberry Pi* pomoći će i slika 1.2. iz poglavlja o *Raspberry Pi Zero W-u ,* gdje su prikazani ulazi i izlazi za generalnu upotrebu, odnosno *GPIO* pinovi.

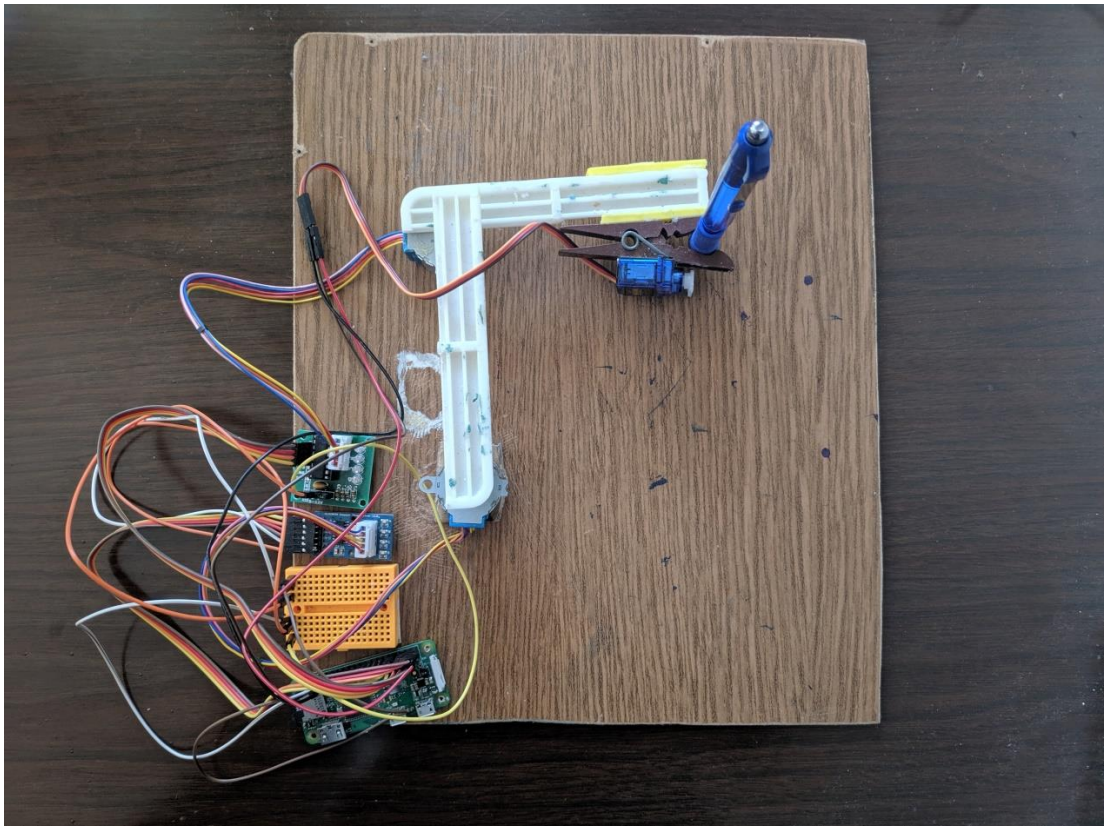


Slika 3.8. Shematski prikaz ožičavanja robotskog manipulatora

Kako sami manipulator ima tri motora kojima treba upravljati, potrebno je dodati dodatnu ploču za ožičavanje komponenata zbog toga što se motori moraju napajati strujom od 5 volti, a sami *Raspberry Pi Zero W* posjeduje samo dva pina za napajanje od 5 volti. Stoga je prvo na dodatnu ploču za ožičavanje potrebno spojiti pin 2 *Raspberry Pi-a*, odnosno pin koji daje napajanje od 5 volti, kako bi se na nju naknadno mogli spajati motori. Također, na dodatnu ploču za ožičavanje se spaja i uzemljenje koje se nalazi na pinu 3 *Raspberry Pi-a* ili na bilo koji pin koji daje uzemljenje.

Budući da se koračni motori ne mogu direktno spajati i upravljati s *Raspberry Pi-om* potrebno ih je spojiti na upravljački sklop za unipolarne motore *ULN2003A.* Nakon što se spoje koračni motori na upravljački sklop potrebno je spojiti sami sklop s *Raspberry Pi-om*. Kao što je već rečeno, sami upravljački sklop ima četiri ulaza koji se spajaju na *GPIO* pinove na *Raspberry P-u*. Upravljački sklopovi za motore imaju označene ulaze *IN1,*

IN2, IN3 i IN4. Prvi koračni motor koji se nalazi u ishodištu koordinatnog sustava manipulatora, odnosno njegov upravljački sklop spaja se redom na *GPIO* pinove: 12, 16, 20 i 21, dok se drugi upravljački sklop za koračne motore spaja na *GPIO* pinove: 6, 13, 19 i 26. Navedeni pinovi odabrani su proizvoljno iz skupa dostupnih *GPIO* pinova. Preostaje samo spojiti napajanje i uzemljenje upravljačkog sklopa s dodatnom pločom za ožičavanje na koju su povezani uzemljenje i napajanje od 5 volti. Za kraj preostaje ožičiti i mikro servo motor *Tower Pro SG90*. Servo motor ima tri ulaza, odnosno tri žice koje je potrebno povezati. Smeđa žica predstavlja uzemljenje koje se spaja na dodatnu ploču gdje je spojeno uzemljenje s *Raspberry Pi-a*. Crvena žica predstavlja napajanje te se ona spaja na dodatnu ploču gdje je spojeno napajanje s *Raspberry Pi-a*. Narančasta žica se spaja direktno na *Raspberry Pi,* odnosno na *GPIO* pin 3 preko kojeg se omogućava upravljanje servo motorom s *Raspberry Pi-om*. Na slici 3.9. prikazan je konačan izgled robotskog manipulatora za ispisivanje fotografija.



Slika 3.9. Robotski manipulator za ispisivanje fotografija

4. PROGRAMSKA PODRŠKA ROBOTSKOG MANIPULATORA

U ovom poglavlju diplomskog rada je opisana programska podrška manipulatora za ispisivanje fotografije. Programska podrška pisana je u programskom jeziku *Python*. Zbog toga je na *Raspberry Pi Zero W* prije svega potrebno instalirati *Python*. *Python* se jednostavno može instalirati preko prozora za upravljanje *Raspberry Pi*-om, kojem se pristupilo preko *SSH* mrežnog protokola, naredbom:

sudo apt-get install python3

Programska podršku manipulatora se dijeli na tri glavna dijela. Prvi dio se odnosi na programsku podršku upravljanja motora robotskog manipulatora te se taj dio programske podrške nalazi na *Raspberry Pi-u*. Osim upravljanja motora potrebno je napraviti i matematički model manipulatora koji se implementira u programsku podršku upravljanja manipulatora kako bi se manipulator mogao gibati u zadanom koordinatnom sustavu po x-y koordinatama. Drugi dio programske podrške manipulatora odnosi se na programsku podršku obrade fotografije. Fotografiju je potrebno obraditi tako da se dobiju točake u koordinatnom sustavu fotografije koje predstavljaju okomite linije po kojima se manipulator giba. Prilikom kretanja po zadanim točkama manipulator ispisuje fotografiju kemijskom olovkom. Također se računaju točke po kojima se manipulator giba kada ne ispisuje okomite linije, odnosno točke kojima manipulator mora proći da bi došao na početnu poziciju za ispis sljedeće linije. Ovaj dio programske podrške odvija se na računalu te se rezultati obrade šalju na *Raspberry Pi*. Na samom kraju je potrebno napraviti dodatnu metodu u programskoj podršci upravljanja robotskog manipulatora koja obrađuje dobivene točke fotografija kako bi se manipulator mogao gibati po tim točkama i na taj način ispisivati fotografiju.

Programska podrška manipulatora i obrade fotografije je pisana objektno orijentiranim programiranjem. Objektno orijentirano programiranje je jedan od najpopularnijih pristupa pisanja programske podrške. Ovaj pristup programiranju pruža mogućnost strukturalnog programiranja tako da se karakteristike i radnje grupiraju u pojedine objekte [10].

4.1. Programska podrška upravljanja robotskog manipulatora

U ovom dijelu je opisana programska podrška upravljanja robotskog manipulatora koja je pisana u programskom jeziku *Python*. Prije svega opisane su dodatne *Python* biblioteke koje se koriste. Nakon toga se stvara razred koji u sebi sadržava sve informacije vezane za manipulator te metode, odnosno radnje koje manipulator ima mogućnost obavljanja. Ovim razredom je omogućeno upravljanje motorima manipulatora preko drugog računala koje je spojeno s *Raspberry Pi*. Osim upravljanja motorima omogućuje se upravljanje manipulatora u x-y koordinatnom sustavu pomoću implementiranog matematičkog modela. Postoje i metode pomoću kojih se može provjeriti je li manipulator uspješno izveden, odnosno je li matematički model ispravno implementiran. Određuje se i najbolje područje rada manipulatora za ispisivanje fotografije.

4.1.1 Python biblioteke za programsku podršku upravljanja robotskim manipulatorom

Prije pisanja programske podrške upravljanja robotskog manipulatora potrebno je na *Raspberry Pi* instalirati sve biblioteke koje se koriste te ih objasniti. Instalacija se vršiti pomoću alata *Python Package Index,* odnosno pomoću *pip3* alata koji služi za instalaciju *Python* biblioteka [11]. Stoga je potrebno u prozoru za upravljanje *Raspberry Pi-om* upisati naredbu kako bi se sami alat instalirao na *Raspberry Pi* :

sudo apt install python-pip3

U nastavku su navedene i opisane korištene biblioteke.

- ***RPi.GPIO***

Ovo je popularna *Python* biblioteka koja se koristi na platformi *Raspberry Pi*-a kako bi se omogućila kontrola *GPIO* pinovima na *Raspberry Pi-u* [12]. U ovom radu se koristi za kontrolu koračnih motora koji imaju mogućnost gibanja u oba smjera. Kako bi instalirali ovu biblioteku potrebno je u prozoru za upravljanje *Raspberry Pi-om* napisati naredbu :

pip3 install RPi.GPIO

- ***pigpio***

Kako je pomoću *RPi .GPIO* biblioteke otežano upravljanje servo motorima, koristi se još jedna biblioteka za upravljanje *GPIO* pinovima na *Raspberry Pi-u,* odnosno biblioteka *pigpio*. Pomoću ove biblioteke se u radu upravlja položajem servo motora preko

PWM(eng. Pulse Width Modulation) metode [13]. Naredbu koju treba izvršiti kako bi se instalirali biblioteka je:

pip3 install pigpio

- ***time***

Ova biblioteka služi za kontrolu vremena unutar *Python* skripte. Jedina funkcija koja se koristi iz ove biblioteke je *sleep()* pomoću koje se odgađa izvedba naredbe za određeno vrijeme koje se funkciji predaje u sekundama [14]. Ova biblioteka je standardna *Python* biblioteka pa je nije potrebno instalirati.

- ***readchar***

Biblioteka *readchar* služi za interakciju računala s *Raspberry Pi-om.* Točnije pomoću nje se omogućuje upravljanje manipulatorom preko tipkovnice računala gdje se pomoću pojedinih tipki mogu pokretati motori te u konačnici kontrolirati manipulator u x-y koordinatnom sustavu [15]. Kako ovo nije standardna *Python* biblioteka potrebno ju je instalirati pomoću naredbe:

pip3 install readchar

- ***math***

Ovo je standardna *Python* biblioteka koja služi za korištenje osnovnih matematičkih funkcija kao što razne trigonometrijske funkcije kojima se izvodi matematički model manipulatora [16]. Instalacija nije potrebna budući da je ovo standardna *Python* biblioteka.

- ***json***

Ova biblioteka služi za pohranu i čitanje podataka u strukturiranom oblik [17]. Pomoću nje se pohranjuju točke koje se manipulatoru predaju kako bi se dogodilo ispisivanje fotografije. Naredba za instalaciju ove biblioteke je :

pip3 install json

Sve ove biblioteke je potrebnom unijeti u *Python* skriptu na slici 4.1. nazvanu *Plotter.py* u koju je implementiran razred manipulatora koji je objašnjena u idućem dijelu.

```
import RPi.GPIO as GPIO
from time import sleep
import readchar
import math
import numpy
import json
import pigpio
```

Slika 4.1. Unos biblioteka korištenih za programsku podršku upravljanja manipulatora

4.1.2. *Python* razred za upravljanje robotskim manipulatorom

Kao što je već rečeno, prilikom izvođenja programske podrške upravljanja manipulatora koristi se metoda objektno orijentiranog programiranja. Stoga je prvo potrebno stvoriti razred u kojem su unutar konstruktora postavljeni svi početni uvjeti. Razred je nazvana *Plotter.*

```
class Plotter():
    def __init__(self):
        ### STEPPER MOTORS
        GPIO.setwarnings(False)
        GPIO.setmode(GPIO.BCM)
        self.motor_channel_2 = (6, 13, 19, 26)
        self.motor_channel_1 = (12, 16, 20, 21)
        GPIO.setup(self.motor_channel_1, GPIO.OUT)
        GPIO.setup(self.motor_channel_2, GPIO.OUT)
        ### SERVO MOTORS
        self.rpi = pigpio.pi()
        self.servo_pin = 3
        self.rpi.set_PWM_frequency(self.servo_pin, 50)
        self.rpi.set_servo_pulsewidth(self.servo_pin, 1500)
```

Slika 4.2. Python kod konstruktora razreda za upravljanje manipulatorom

U kodu prikazanom na slici 4.2. se može vidjeti da se prvo provodi inicijalizacija samih motora. Prvo se vrši inicijalizacija koračnih motora preko *Python* biblioteke *RPI.GPIO* koja je učitana u *Python* skriptu kao *GPIO*. Pozivanjem funkcije *GPIO.setmode(flag),* koja se nalazi u *Python* paketu *RPi.GPIO,* se dobiva pristup *GPIO* pinovima *Raspberry Pi-a.* Ulaz u ovu funkciju je zastavica *GPIO.BCM*[12]. Ovaj ulaz predstavlja inicijalizaciju pinova numerički jednaku kao kod prikaza *GPIO* pinova na slici 1.2.. Nadalje je potrebno postaviti dvije n-torke koje predstavljaju *GPIO* pinove koračnih motora. Kao što je u

poglavlju o ožičavanju rečeno, prvi koračni motor se spaja na *GPIO* pinove: 12,16, 20 i 21 pa se tako i n-torka prvog koračnog motora postavlja u te iste vrijednosti. N-torka za prvi koračni motor je pohranjena kao varijabla instance i nazvana je *self.motor_channal_1.* Analogno se za drugi koračni postavlja n-torka s *GPIO* pinovima: 6, 13, 19 i 26. Nakon postavljanja koračnih motora u određene pinove *Raspberry Pi-a* potrebno je pozvati funkciju *GPIO.setup(gpio_pin , flag)* koja ima dva ulaza. Prvi ulaz predstavlja n-torku s *GPIO* pinovima na koje su koračni motori spojeni, dok drugi ulaz predstavlja zastavicu koja označava jesu li pinovi koji su navedeni ulazi ili izlazi [12]. Tako se *GPIO* pinovi. koji su postavljeni za koračne motore. inicijaliziraju kao izlazi budući da je potrebno slati signale pinovima. Analogno je potrebno pozvati istu funkciju i za drugi koračni motor.

U idućem dijelu koda se omogućuje pristup mikro servo motoru. Kao što je već rečeno za servo motor se koristi druga biblioteka koja omogućava pristup *GPIO* pinovima *Raspberry Pi-a*, a to je *Python* biblioteka *pigpio.* Pristup *GPIO* pinovima preko *pigpio* biblioteke omogućuje se s funkcijom unutar biblioteke koja se zove *pigpio.pi()* te se stvori novi objekt, pohranjen u varijablu instance *self.rpi*, koji omogućuje pristup određenim pinovima *Raspberry Pi-a* [13]. Servo motor je povezan s *GPIO* pinom 3 pa je stoga sama varijabla instance *self.servo_pin* postavljena u broj 3. Razlog zbog koje koristimo *pigpio* biblioteku umjesto *RPi.GPIO* biblioteke je taj što ova biblioteka ima bolji pristup upravljanju motorima preko pulsno širinske modulacije signala *(eng. Pulse Width Modulation)* ili skraćeno *PWM*. Ova metoda radi na način da stvara prosječnu veličinu predanog električnog signala na način da ga reže na diskretne vrijednosti, odnosno periode na određenoj frekvenciji . Sami periodi signala označavaju koji položaj servo motor zauzima pri određenom periodu. Frekvencija mikro servo motora *SG90* jednaka je 50Hz, dok je raspon *PWM* perioda 20ms. *PWM* period signala od 1ms predstavlja položaj motora na 0°, 1.5ms predstavlja položaj motora na 90°, dok 2ms predstavlja položaj motora na 180° [9]. Kao što je već rečeno ovaj servo motor radi na frekvenciji od 50Hz te je stoga potrebno pozvati funkciju *self.rpi.set_PWM_frequency(user_gpio, frequency)* kako bi postavili frekvenciju rada na kojoj servo motor radi. Prvi ulaz u funkciju predstavlja već definirana varijabla instance *GPIO* pina servo motora, dok je drugi ulaz frekvencija na kojoj servo motor radi [13]. U idućem redu koda postavljamo servo motor u početni položaj funkcijom *self.rpi.set_servo_pulsewidth(user_gpio, pulsewidth)* gdje su ulazi u funkciju varijabla

instance *GPIO* pina servo motora te *PWM* period, odnosno kut na koji postavljamo servo motor. Vrijednost *PWM* perioda je 1500, odnosno 1.5 ms. Tako je početni kut na koji postavljamo servo motor 90°.

Nakon što se uspješno pristupilo *GPIO* pinovima na koji su motori spojeni potrebno je definirati metode unutar razreda za upravljanje manipulatorom. Prije svega je potrebno napraviti novu varijablu instance *self.delay* koja predstavlja kašnjenje motora. Kašnjenje se postavlja u vrijednost decimalnog broja 0.01. Vrijednost kašnjenja je definirana u sekundama. Ovom vrijednosti se određuje brzina rada samog manipulatora, odnosno brzina pokretanja koračnih motora. Samo kašnjenje se može postaviti i na manju vrijednost, ali zbog prevelike brzine pomicanja ruku manipulatora dobivaju se lošiji rezultati ispisa.

```
class Plotter():
        ---odrezano---
    def move_clockwise_1(self):
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.LOW, GPIO.LOW, GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.LOW, GPIO.HIGH, GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH, GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW, GPIO.LOW))
        sleep(self.delay)
    def move_clockwise_2(self):
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.LOW, GPIO.LOW, GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW, GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH, GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.LOW, GPIO.HIGH, GPIO.HIGH))
        sleep(self.delay)
    def move_anti_clockwise_1(self):
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.LOW, GPIO.LOW, GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW, GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH, GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.LOW, GPIO.HIGH, GPIO.HIGH))
        sleep(self.delay)
    def move_anti_clockwise_2(self):
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.LOW, GPIO.LOW, GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.LOW, GPIO.HIGH, GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH, GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW, GPIO.LOW))
        sleep(self.delay)
```

Slika 4.3. *Python* kod metode za pokretanje koračnih motora

U kodu prikazanom na slici 4.3. su prikazane četiri metode unutar razreda *Plotter* koje služe za pokretanje koračnih motora. Prva metoda, odnosno metoda *move_clockwise_1*(*self)* predstavlja pokretanje prvog koračnog motora u smjeru kazaljke

na satu. Unutar ove metode se vrši slanje signala pinovima na *Raspberry Pi* određenim sekvencijskim redom. Nadalje se taj signal šalje do zavojnica unutar koračnog motora koje se pomoću ove metode okidaju. Funkcija za slanje signala pinovima uz pomoć biblioteke *RPi.GPIO* je *GPIO.output(gpio_pins, pin_status)*[12]. Sama funkcija ima dva ulaza od kojih prvi ulaz predstavlja pinove koračnog motora, odnosno varijablu u obliku n-torke *self.motor_channal_1.* Kod unipolarnog koračnog motora *28-BYJ-48* postoje četiri faze, odnosno četiri zavojnice pa je zbog toga potrebno napraviti četiri koraka okidanja faza pri kojima koračni motor napravi jedan korak. Stoga je drugi ulaz n-torka s četiri člana pomoću koje se označava stanje pinova preko zastavice. Vrijednost zastavice *GPIO.HIGH* predstavlja slanje signala datom pin, a vrijednost *GPIO.LOW* označava da se signal ne šalje. Tako je sekvenca za okidanja pinova kada želimo da se prvi koračni motor pokreće za jedan korak u smjeru kazaljke na satu prikazan u tablici 4.1..

FAZE	*GPIO pin 12*	*GPIO pin 16*	*GPIO pin 20*	*GPIO pin 21*
1.korak	*GPIO.HIGH*	*GPIO.LOW*	*GPIO.LOW*	*GPIO.HIGH*
2.korak	*GPIO.HIGH*	*GPIO.HIGH*	*GPIO.LOW*	*GPIO.LOW*
3.korak	*GPIO.LOW*	*GPIO.HIGH*	*GPIO.HIGH*	*GPIO.LOW*
4.korak	*GPIO.LOW*	*GPIO.LOW*	*GPIO.HIGH*	*GPIO.HIGH*

Tablica 4.1. Sekvencijski red slanja signala za pokretanje prvog koračnog motora u smjeru kazaljke na satu

Metoda za pokretanje drugog koračnog motora se zove *move_clockwise_2*(*self).* Analogno prvom koračnom motoru sekvenca pokretanja motora je jednaka kao kod prvog koračnog motora i prikazana je u tablici 4.2..

FAZE	*GPIO pin 6*	*GPIO pin 13*	*GPIO pin 19*	*GPIO pin 26*
1.korak	*GPIO.HIGH*	*GPIO.LOW*	*GPIO.LOW*	*GPIO.HIGH*
2.korak	*GPIO.HIGH*	*GPIO.HIGH*	*GPIO.LOW*	*GPIO.LOW*
3.korak	*GPIO.LOW*	*GPIO.HIGH*	*GPIO.HIGH*	*GPIO.LOW*
4.korak	*GPIO.LOW*	*GPIO.LOW*	*GPIO.HIGH*	*GPIO.HIGH*

Tablica 4.2. Sekvencijski red slanja signala za pokretanje prvog koračnog motora u smjeru kazaljke na satu

Iduća metoda služi za pokretanje koračnih motora u smjeru obrnutom od kazaljke na satu. Metoda za prvi koračni motor je nazvana *move_anti_clockwise_1(self)* , dok je za drugi koračni motor nazvana *move_anti_clockwise_2(self).* Sekvenca okidanja zavojnica koračnih motora, odnosno slanje signala pinova *Raspberry Pi*-a kada se motori rotiraju u smjeru obrnutom od kazaljke na satu je prikazana unutra tablice 4.3..

FAZE	*GPIO pin 6* *GPIO pin 12*	*GPIO pin 13* *GPIO pin 16*	*GPIO pin 19* *GPIO pin 20*	*GPIO pin 26* *GPIO pin 21*
1.korak	*GPIO.HIGH*	*GPIO.LOW*	*GPIO.LOW*	*GPIO.HIGH*
2.korak	*GPIO.LOW*	*GPIO.LOW*	*GPIO.HIGH*	*GPIO.HIGH*
3.korak	*GPIO.LOW*	*GPIO.HIGH*	*GPIO.HIGH*	*GPIO.LOW*
4.korak	*GPIO.HIGH*	*GPIO.HIGH*	*GPIO.LOW*	*GPIO.LOW*

Tablica 4.3. Sekvencijski red slanja signala za pokretanje prvog i drugog koračnog motora u smjeru obrnutom od kazaljke na satu

Također, unutar razreda postoje i četiri metode pomoću kojih se oba motora istovremeno pomiču u smjeru koji se traži. Ove metode su prikazane u prilogu gdje se nalazi cijeli kod programske podrške. Tako metoda *move_anti1_clock2(self)* služi za pokretanja prvog koračnog motora u smjeru obrnutom od smjera kazaljke na satu, dok se drugi koračni motor pokreće u smjeru kazaljke na satu. Analogno po imenu metode se može zaključiti što ostale metode rade, a one su: *move_clock1_anti2(self), move_anti_clockwise_12(self) i move_clocwise_12(self).*

Nakon što su objašnjene metode za pokretanje koračnih motora potrebno je objasniti metode za pomicanje servo motora. Postoje dvije metode za pokretanje servo motora koje su prikazane na slici 4.4.. Pomoću prve metode *pen_up(self)* se položaj motora postavlja u položaj kada servo motor podiže kemijsku olovku od radne ploče crtača, dok je druga metoda *pen_down(self)* služi za postavljanje servo motora u položaj kada kemijska olovka dodiruje površinu ploče crtača, odnosno papir po kojem će se ispisivati fotografija.

```
class Plotter():
        ---odrezano---
    def pen_up(self):
        self.rpi.set_servo_pulsewidth(self.servo_pin, 1500)
        sleep(0.1)
    def pen_down(self):
        self.rpi.set_servo_pulsewidth(self.servo_pin, 1000)
        sleep(0.1)
```

Slika 4.4. *Python* kod metoda za pokretanje servo motora

Metodom *pen_up(self)* se servo motor postavlja na kut od 90°, odnosno plastični nastavak na vratilu rotora servo motora u kojem je kemijska olovka odmaknuta od podloge manipulatora. S druge strane metoda *pen_down(self)* postavlja plastični nastavak na vratilu rotora motora u položaj u kojem kemijska olovka dira podlogu manipulatora i na taj način omogućen je ispis po papiru. Unutar obje funkcije poziva se funkcija *self.rpi.set_servo_pulsewidth(user_gpio, pulsewidth)* iz biblioteke *pigpio* pomoću koje postavljamo servo motor u određeni položaj. Ulaz u funkciju je varijabla instance *self.servo_pin* koja označava *GPIO* pin na koji je spojen servo motor, dok drugi ulaz predstavlja *PWM* period, za položaj u kojem je kemijska olovka podignuta, postavljen u 1500(1.5ms) te tada servo motor zauzima kut od0 90°. Kada je kemijska olovka spuštena, period je postavljen na 1000(1ms) što znači da se motor zarotira na položaj od 0° te se u tom trenutku plastični nastavak na vratilu rotora motora postavi u položaj u kojem se omogući dodir kemijske olovke s radnom površinom manipulatora.

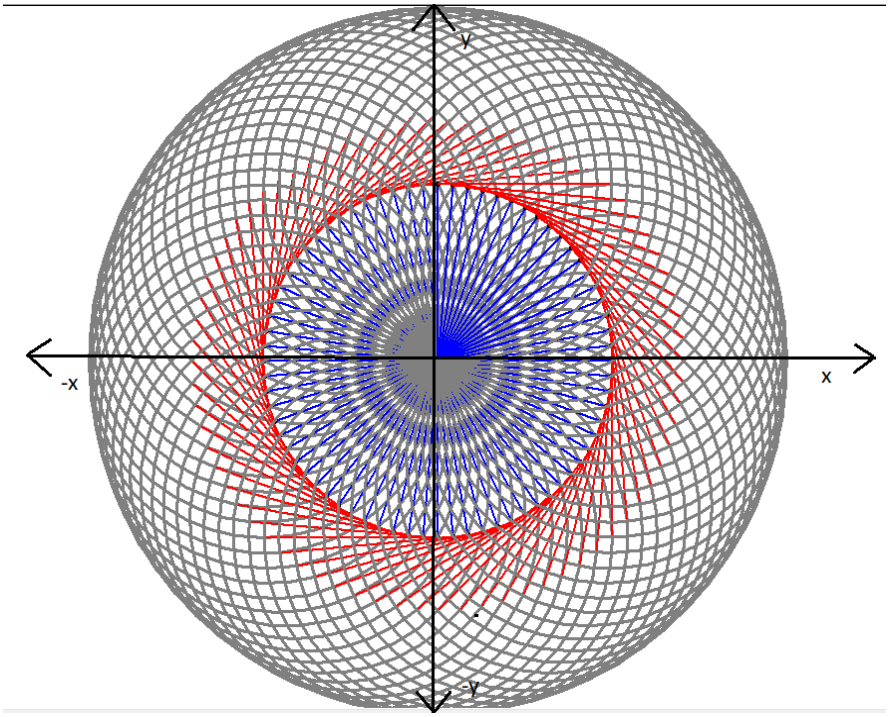
Kako bi provjerili funkcioniraju li definirane metode, definirana je metoda *move_to_start_point(self)*(slika 4.5.) pomoću koje možemo upravljati motorima, odnosno pokretati ruke manipulatora te podizati i spuštati kemijsku olovku od podloge manipulatora. Funkcija ima ovaj naziv zato što je sami manipulator prije ispisa potrebno postaviti u početni položaj, što će biti objašnjeno kasnije.

```
class Plotter():
      ---odrezano---
      def move_to_start_point(self):
          while True:
              key = readchar.readchar()

              if key == "e":
                  self.move_clockwise_1()
                  print('Moving stepper_1 anticlockwise')
              elif key == "w":
                  self.move_anti_clockwise_1()
                  print('Moving stepper_1 clockwise')
              elif key == "s":
                  self.move_anti_clockwise_2()
                  print('Moving stepper_2 anticlockwise')
              elif key == "d":
                  self.move_clockwise_2()
                  print('Moving stepper_2 clockwise')
              elif key == 't':
                  self.pen_up()
                  print('Pen up!')
              elif key =='g':
                  self.pen_down()
                  print('Pen down!')
              elif key == "q":
                  break
```

Slika 4.5. *Python* kod metode za pokretanje motora s tipkovnicom računala

Unutar metode postavlja se *while* petlja koja konstantno provjerava pritisnute tipke na tipkovnici računala. U *while* petlji se postavlja varijabla koja pomoću funkcije *readchar.readchar(),* iz *Python* biblioteke *readchar,* postavlja pritisnutu tipku u varijablu *key* [15]. Nadalje se pomoću *if* petlje provjerava koja je tipka pritisnuta. Tako se pritiskanjem tipke *'e'* na tipkovnici računala prvi koračni motor pomiče u smjeru kazaljke na satu, odnosno poziva se metoda *self.move_clockwise_1()*, pritiskanjem tipke '*w*' se prvi motor pomiče u smjeru obrnutom od kazaljke na satu, odnosno poziva se metoda *self.move_anti_clockwise_1()* . Analogno se provjeravaju i tipke za pokretanje drugog motor te se pozivaju metode za pokretanje drugog motora. Za pomicanje drugog motora u smjeru kazaljke na satu odabrana je tipka '*d*', dok je za smjer obrnut od kazaljke na satu odabrana tipka *'s'* na tipkovnici računala. Također se provjeravaju i tipke za pomicanje servo motora u gornji i donji položaj kemijske olovke. Tipka '*t*' služi za postavljanje servo motora tako da je kemijska olovka podignuta od podloge, odnosno pritiskanjem tipke se poziva metoda *pen_up(),* dok je tipka '*g*' za spuštanje kemijske olovke, odnosno poziva se metoda *pen_down()*. Ako želimo prekinuti petlju potrebno je pritisnuti tipku *'q'*. Sa svim ovim metodama unutar razreda *Plotter* je omogućeno

pokretanje manipulatora bez ikakvih ograničenja i matematičkog modela koji definira položaj mehanizma manipulatora u koordinatnom sustavu. Na slici 4.6. je prikazano područje ispisa samog manipulatora kada su koračni motori postavljeni u x-y koordinatni sustav u kojem prvi koračni motor predstavlja ishodište koordinatnog sustava. Budući da se koračni motori mogu okretati za 360°, potrebno je napraviti matematički model koji će ograničiti područje gibanja manipulatora s obzirom na konstrukciju manipulatora i željeno područje crtanja. Implementacijom matematičkog modela omogućit će se pokretanja mehanizma manipulatora u x-y koordinatnom sustavu te će se znati položaj kuta koji motori moraju zauzeti za taj položaj. Na slici 4.6. crvenom bojom označen je položaj vanjske ruke manipulatora, dok je plavom bojom označen položaj koji zauzima unutarnja ruka manipulatora za određene kuteve koji motori zauzimaju prilikom pomicanja. Siva boja označava područje po kojoj kemijska olovka može crtati. Iscrtavanje područja rada je napravljeno u *Pythonu* pomoću biblioteke *turtle* koja omogućava vizualizacije simulacija. Kod simulacije kretanja ruku manipulatora se nalazi u prilogu. Iz rezultata simulacije se može vidjeti da je područje rada manipulatora jednako u svakom kvadrantu koordinatnog sustava manipulatora.



Slika 4.6. Područje rada manipulatora bez matematičkog modela i mehaničkih ograničenja

4.2. Matematički model mehanizma manipulatora

U ovom potpoglavlju je raspisan matematički model kojim se omogućuje gibanje manipulatora u x-y smjeru koordinatnog sustava te se model implementira u programsku podršku upravljanja manipulatora. Cilj matematičkog modela je postići da manipulator postavlja kemijsku olovku u dati položaj u x-y koordinatnom sustavu gdje je prvi koračni motor predstavlja samo ishodište. Pomoću matematičkog modela omogućuje se kretanje kemijske olovke u pozitivnom i negativnom smjeru osi x, te u pozitivnom i negativnom smjeru osi y. Kretanje se omogućuje na način da se računaju kutevi koračnih motora koje manipulator treba zauzeti u odnosu na početni kut koračnog motora, koji se definira za početni položaj manipulatora, kako bi se dobio željeni položaj. Prije raspisivanja matematičkog modela potrebno je postaviti ograničenja koju sama mehanika crtača traži. Također je potrebno odrediti početni položaj kemijske olovke u odnosu na ishodište koordinatnog sustava. Početni položaj ruku potrebno je definirati zbog toga što koračni motori nemaju povratnu vezu te se zbog toga ne zna njihov točan položaj nakon pomicanja. Programskim kodom se omogućuje računanje i pamćenje položaja koračnih, odnosno kuta kojeg ruke manipulatora zauzimaju nakon pomicanja u željenu poziciju. Matematički model se temelji na kosinusovom poučku i standardnim trigonometrijskim funkcijama.

4.2.1. Kosinusov poučak

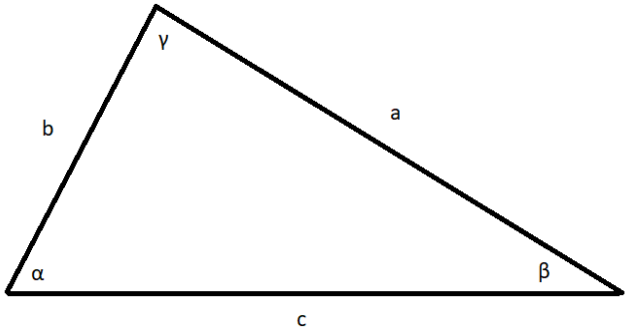
Prije nego što se izvede matematički model potrebno je raspisati formule vezane za kosinusov poučak koje se koriste za izvod. Kosinusov poučak se danas još naziv i *Cartonov* poučak jer ga je on prikazao u sadašnjem obliku [18]. Kosinusov poučak je matematički izraz pomoću kojeg je moguće računati relacije između elemenata trokuta (slika 4.7.) [18].

Definicija kosinusovog poučka glasi da je kvadrat duljine jedne stranice trokuta jednak zbroju kvadrata duljina drugih dviju stranica, umanjenom za dvostruki produkt duljina tih stranica i kosinusa kuta kojeg one određuju [18]. Za trokut na slici 4.7. formule glase:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha \quad (1)$$

$$b^2 = a^2 + c^2 - 2ac\cos\beta \quad (2)$$

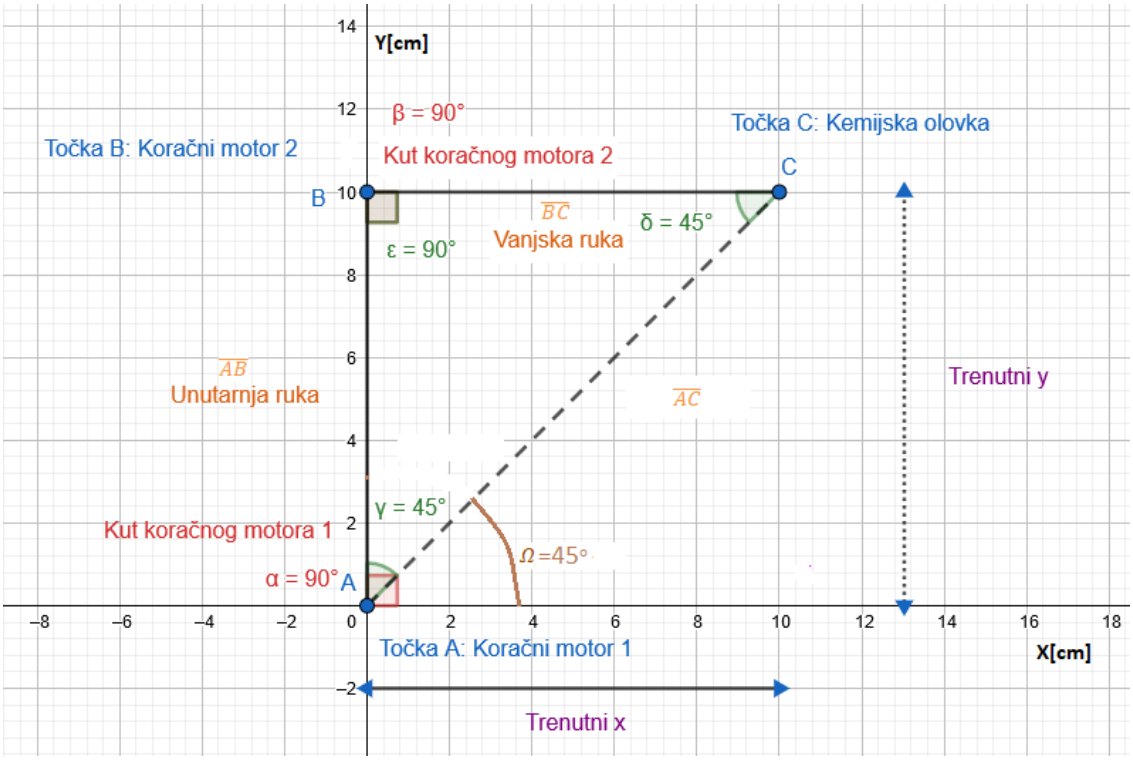
$$c^2 = a^2 + b^2 - 2ab\cos\gamma \quad (3)$$



Slika 4.7. Kosinusov poučak

4.2.2. Izvod matematičkog modela

Na slici 4.8. je prikazan pojednostavljeni prikaz početnog položaja mehanizma manipulatora. Zbog same konstrukcije manipulatora, područje po kojem se ruke manipulatora mogu nesmetano gibati je ono kada se kemijska olovka nalazi unutar prvog i četvrtog kvadranta koordinatnog sustava manipulatora . Stoga se matematički model raspisuje tako da kemijska olovka može zauzimati položaj u samo u ta dva kvadranta.



Slika 4.8. Pojednostavljeni prikaz početnog položaja mehanizma manipulatora

Na slici 4.8. točka A predstavlja ishodište koordinatnog sustava, odnosno položaj prvog koračnog motora. Točka B predstavlja zglob između unutarnje ruke manipulatora i vanjske ruke manipulatora, gdje se nalazi drugi koračni motor. Kemijsku olovku na slici predstavlja točka C. Između točaka A i B nalazi se unutarnja ruka manipulatora, dok se između točke B i C nalazi vanjska ruka manipulatora. Dužina vanjske i unutarnje ruke manipulatora je oko 10 centimetara. Tako se pomoću slike može definirati početni položaj kemijske olovke u x-y koordinatnom sustavu:

$$\text{trenutni\_x} = \overline{BC} = \sim 10\text{cm} \tag{4}$$

$$\text{trenutni\_y} = \overline{AB} = \sim 10\text{cm} \tag{5}$$

Nadalje je potrebno definirati početne kuteve koje ruke manipulatora zatvaraju u odnosu na koordinatni sustav gdje kut α_0 predstavlja kut prvog koračnog motora, odnosno unutarnje ruke manipulatora u odnosu na pozitivnu os x, dok kut β_0 predstavlja kut koji vanjska ruka zatvara s unutarnjom rukom manipulatora.

$$\alpha_0 = \beta_0 = 90° \tag{6}$$

Sami matematički model se temelji na kosinusovom poučku. Stoga je, kao što se vidi na slici, potrebno definirati najbližu udaljenost između ishodišta koordinatnog sustava do položaja kemijske olovke kako bi se dobio zatvoreni trokut. Najbliža udaljenost ustvari predstavlja hipotenuzu trokuta mehanizma te je njen iznos definiran formulom:

$$\overline{AC} = \sqrt{(trenutni\_x)^2 + (trenutni\ y)^2} \tag{7}$$

Kako bi bilo moguće primijeniti kosinusov poučak, potrebno je definirati i početne kuteve trokuta mehanizma:

$$\gamma_0 = 45° \tag{8}$$

$$\varepsilon_0 = 90° \tag{9}$$

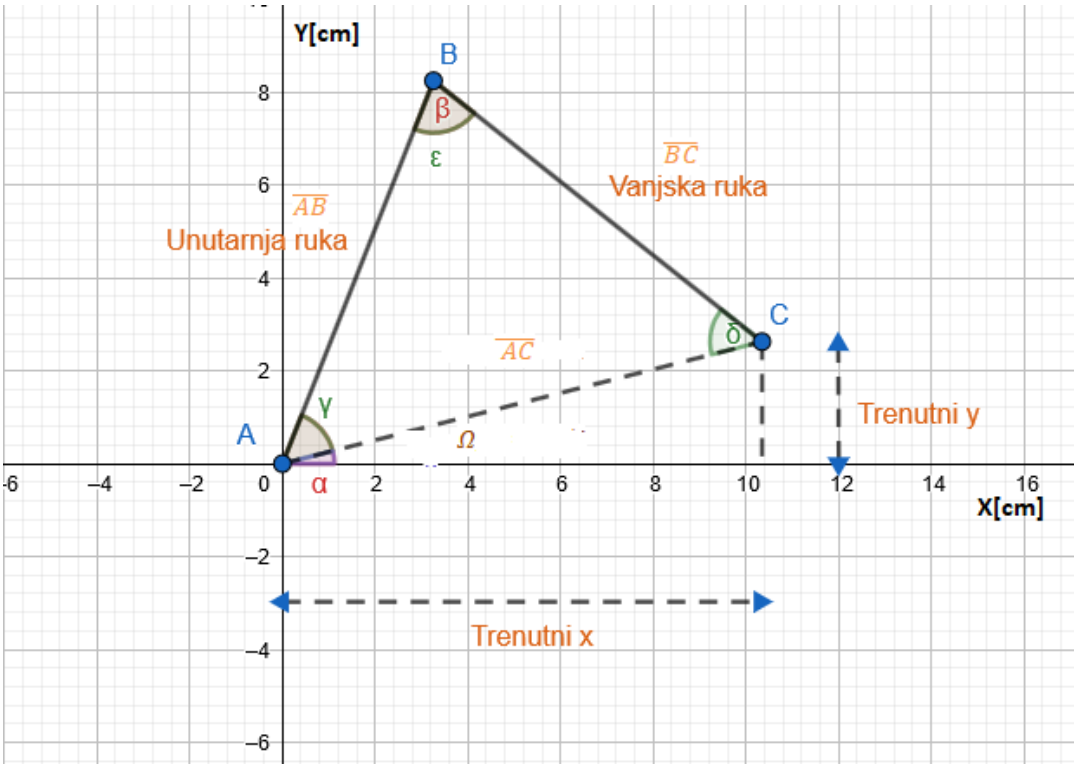
$$\delta_0 = 45° \tag{10}$$

gdje je kut γ_0, smješten u točki A i predstavlja kut koji unutarnja ruka manipulatora zatvara s hipotenuzom trokuta $\overline{AC}$, kut ε_0, smješten u točki B je jednak kutu drugog koračnog motora te on predstavlja kut koji vanjska ruka manipulatora zatvara s unutarnjom rukom i preostaje kut δ_0 koji zatvara kut između vanjske ruke manipulatora

i hipotenuze trokuta $\overline{AC}$ i on se nalazi se u točki C. Za kraj je još potrebno definirati kut koji hipotenuza trokuta, odnosno dužina $\overline{AC}$, zatvara s pozitivnom osi x:

$$\Omega = 45° \tag{11}$$

Kako bi se izveo matematički model, mehanizam manipulatora se postavlja u položaj u kojem je kemijska olovka odmaknuta od početnog položaja (slika 4.9.).



Slika 4.9. Novi položaj mehanizma manipulatora

Cilj je dobiti formulu za izračun vrijednost kuta prvog koračnog motora α, odnosno kut kojeg unutarnja ruka manipulatora $\overline{AB}$ zatvara s osi x u pozitivnom smjeru te kut drugog koračnog motora β, odnosno kut koji vanjska ruka manipulatora $\overline{BC}$ zatvara s unutarnjom rukom manipulatora.

Budući da je kut Ω, odnosno kut koji dužina $\overline{AC}$ zatvara s pozitivnom osi x koordinatnog sustava ključan za razradu modela, prvo je potrebno odrediti njegovu vrijednost. Ona se može izračunati pomoću inverzne trigonometrijske funkcije tangensa ako mu se preda željeni novi položaj kemijske olovke u x-y koordinatama, odnosno omjer između trenutni_y i trenutni_x.

$$\Omega = \tan^{-1}\left(\frac{trenutni\_y}{trenutni\_x}\right) \tag{12}$$

Nakon dobivanja kuta Ω moguće je definirati najbližu udaljenost kemijske olovke od ishodišta koordinatnog sustava $\overline{AC}$ pomoću trigonometrijske funkcije kosinus:

$$\overline{AC} = \frac{trenutni\_x}{\cos\Omega} \tag{13}$$

Kada se dobije dužina $\overline{AC}$ potrebno je izračunati kut γ koji će nam kasnije omogućiti dobivanje kuta α koji je tražen. Kut γ je kut koji se nalazi u trokutu koji zatvara mehanizam manipulatora te se nalazi u točki A. Za izračun kuta koristi se jedna od formula kosinusovog poučka prebačena u oblik za računanje kuta:

$$\gamma = \cos^{-1}\left(\frac{\overline{AB}^2+\overline{AC}^2-\overline{BC}^2}{2\overline{AB}\ \overline{AC}}\right) \tag{14}$$

Nakon dobivanja kuta u točki A moguće je definirati željeni kut α koji unutarnja ruka zatvara s pozitivnom osi x iz jednadžbi (13) i (15):

$$\alpha = \cos^{-1}\left(\frac{\overline{AC}^2+\overline{AB}^2-\overline{BC}^2}{2\overline{AB}\ \overline{AC}}\right) + \tan^{-1}\left(\frac{trenutni\_y}{trenutni\_x}\right) \tag{15}$$

Još je preostalo definirati jednadžbu za računanje kuta dugog koračnog motora, odnosno kuta β koji unutarnja ruka zatvara s vanjskom rukom manipulatora. Opet se koristiti jedna od jednadžbi za računanje kuta, u točki B koju zatvara mehanizam manipulatora, preko kosinusovog poučaka :

$$\varepsilon = \cos^{-1}\left(\frac{\overline{AB}^2+\overline{BC}^2-\overline{AC}^2}{2\overline{AB}\ \overline{BC}}\right) \tag{16}$$

Budući da je kut trokuta u točki B jednak kutu drugog koračnog motora, može se pisati :

$$\beta = \varepsilon = \cos^{-1}\left(\frac{\overline{AB}^2+\overline{BC}^2-\overline{AC}^2}{2\overline{AB}\ \overline{BC}}\right) \tag{17}$$

Iz matematičkog modela se može zaključiti da kut β, odnosno kut koji vanjska ruka manipulatora zatvara s unutarnjom rukom manipulatora mora biti veći od 0° i manji od 180° kako bi mehanizam uvijek zatvarao trokut pomoću kojeg je izveden matematički model.

4.3. Implementacija matematičkog modela u programsku podršku upravljana manipulatora

Nakon izvođenja matematičkog modela, potrebno ga implementirati u programsku podršku upravljanja manipulatorom. Unutar razreda *Plotter* definira se nova metoda *get_position_xy(self)* koja u sebi sadrži matematički model koji je u prošlom poglavlju raspisan. Pomoću ove metode postiže se mogućnost pokretanja mehanizma manipulatora u x-y koordinatama gdje je prvi koračni motor ishodište x-y sustava. Unutar konstruktora razreda inicijaliziraju se varijable instanci iz matematičkog modela definirane u potpoglavlju 4.2.. Unutar *Python* koda (slika 4.10.) su objašnjenja kakvo značenje varijable instanci definirane u konstruktoru razreda imaju u odnosu na matematički model.

```
class Plotter():
    def __init__(self):
        ---odrezano---
        ### Početni položaj manipulatora
        self.arm_1 = 9.2   # Stvaran dužina unutarnje ruke izmjerena na
konstrukciji[cm]
        self.arm_2 = 10.5 # Stvarna dužina vanjske ruke izmjerena na konstrukciji[cm]
        self.current_x = 10.5   # Trenutni x prema matematičkom modelu[cm]
        self.current_y = 9.2   # Trenutni y prema matematičkom modelu[cm]
        self.alfa = 90  # Trenutni/početni kut prvog koračnog motroa
        self.beta = 90  # Trenutni/početni kut drugog koračnog motroa
        #Stupanj za koji se koračni motor rotira za jedan korak
        self.step_angle = 0.703125
        # Greška koju manipulator stvara zbog prevelikog stupnja pomaka koračnog
motra
        self.error1 = 0  # Greška za prvi motor
        self.error2 = 0  # Greška za drugi motor
        # Brojač broja koraka koje manipulatori naprave tokom ispisa fotografije
        self.final_steps_count1 = 0  # Ukupan broj koraka prvog koračnog motora
        self.final_steps_count2 = 0  # Ukupan broj koraka drugog koračnog motora
        # Broj koraka koje manipulator napravi da dođe iz jednog položaja u drugi
        self.round_steps_count1 = 0  # Broj koraka prvog koračnog motora
        self.round_steps_count2 = 0 # Broj koraka drugog koračnog motora
```

Slika 4.10. *Python* kod definiranja početnih uvjeta manipulatora

```
class Plotter():
    ---odrezano---
    def get_position_xy(self):
    ###IMPLEMENTACIJA MATEMATIČKOG MODELA
        omega_x = math.atan(self.current_y / self.current_x)
        AC = self.current_x / math.cos(omega_x)
        gama = math.acos((AC ** 2 + self.arm_1 ** 2 - self.arm_2 ** 2) / (2 *
self.arm_1 * AC))
        new_alfa = omega_x +  gama
        new_alfa_degrees = math.degrees(new_alfa)
        epsilon = math.acos((self.arm_1 ** 2 + self.arm_2 ** 2 - AC ** 2) / (2 *
self.arm_1 * self.arm_2))
        new_beta = epsilon
        new_beta_degrees = math.degrees(new_beta)
    ### RAČUNAJE BROJA KORAKA KOJE MOTOR TREBA NAPRAVIT DA ZAUZME NOVI KUT
        delta_1 = self.alfa - new_alfa_degrees
        self.alfa = new_alfa_degrees
        delta_2 = self.beta - new_beta_degrees
        self.beta = new_beta_degrees
        print('Stepper_angle1: ', self.alfa, ' degrees', 'Stepper_angle2: ',
            self.beta, ' degrees')
        count_steps1 = delta_1 / self.step_angle
        count_steps2 = delta_2 / self.step_angle
        count_steps1 += self.error1
        count_steps2 += self.error2
        self.round_steps_count1 = round(count_steps1)
        self.round_steps_count2= round(count_steps2)
        self.final_steps_count1 += self.round_steps_count1
        self.final_steps_count2 += self.round_steps_count2
        self.error1 = count_steps1 - self.round_steps_count1
        self.error2 = count_steps2 - self.round_steps_count2
        print(self.round_steps_count1,self.round_steps_count2)
    #POKRETANJE MOTORA
        self.move_motors_by_step_count()
```

Slika 4.11. *Python* kod metode za upravljanje motora preko matematičkog modela

Unutar metode *get_position_xy(self)* (slika 4.11.) prvo se implementira matematički model koji za tražene x-y koordinate računa na koji je kut potrebno postaviti koračni motori, odnosno ruke manipulatora. Nakon dobivanja traženih kuteva, računa se razlika između novih kuteva i starih kuteva te se dobivene vrijednosti spremaju u varijable *delta_1 i delta_2*. Kut koji koračni motor prijeđe kada napravi jedan korak definiran je u varijabli instance *self.step_angle* i iznosi 0.703125° kako je izračunato u poglavlju 2.2.. Ako se promjena kuteva podijeli s iznosom kuta za jedan korak, dobiva se broj koraka koji koračni motor mora napraviti kako bi zauzeo traženi položaj te se izračunate vrijednosti spremaju u varijable *count_steps_1 i count_steps2.* Ovom broju koraka se dodaje greška prošlog proračuna koja je definirana varijablama instance za prvi i drugi motor kao *self.error1 i self.error2* te u početku iznose 0. Kako izračunati broj koraka može dati decimalan broj, a motor može raditi samo cijeli korak, dobiveni broj koraka

potrebno je zaokružiti na najbliži cijeli broj te zbog toga nastaje greška. Grešku računamo tako da se broj koraka dobiven u decimalnom broju oduzme od zaokruženog broja koraka. Zaokruženi koraci se spremaju u varijable instance *self.round_steps_count1 i self.round_steps_count2* .Također, unutar metode se definiraju dvije varijable instance koje će pamtiti koliko je pojedini motor napravio koraka tijekom ispisivanja. Ove dvije varijable instance su nazvane *self.final_step_count1* i *self.final_step_count2* i koriste se da bi se, na kraju ispisa fotografije, motor mogao vratiti u početni položaj. Unutar metode se na samom kraju poziva metoda *self.move_motors_by_step_count()* koja za izračunati broj koraka pomiče koračne motore.

```
class Plotter():
        ---odrezano---
      def move_motors_by_step_count(self):
          while self.round_steps_count1 != 0 or self.round_steps_count2 != 0:
              if self.round_steps_count1 >= 1 and self.round_steps_count2 >= 1:
                  self.move_clocwise_12()
                  self.round_steps_count1 -= 1
                  self.round_steps_count2 -= 1
                  print(self.round_steps_count1,self.round_steps_count2)
              elif self.round_steps_count1 <= -1 and self.round_steps_count2 >= 1:
                  self.move_anti1_clock2()
                  self.round_steps_count1 += 1
                  self.round_steps_count2 -= 1
                  print(self.round_steps_count1, self.round_steps_count2)
              elif self.round_steps_count1 <= -1 and self.round_steps_count2 <= -1:
                  self.move_anti_clockwise_12()
                  self.round_steps_count1 += 1
                  self.round_steps_count2 += 1
                  print(self.round_steps_count1, self.round_steps_count2)
              elif self.round_steps_count1 >= 1 and self.round_steps_count2 <= -1:
                  self.move_clock1_anti2()
                  self.round_steps_count1 -= 1
                  self.round_steps_count2 += 1
                  print(self.round_steps_count1, self.round_steps_count2)
              elif self.round_steps_count1 >= 1:
                  self.move_clockwise_1()
                  self.round_steps_count1 -= 1
                  print(self.round_steps_count1, self.round_steps_count2)
              elif self.round_steps_count1 <= -1:
                  self.move_anti_clockwise_1()
                  self.round_steps_count1 += 1
                  print(self.round_steps_count1, self.round_steps_count2)
              elif self.round_steps_count2 >= 1:
                  self.move_clockwise_2()
                  self.round_steps_count2 -= 1
                  print(self.round_steps_count1, self.round_steps_count2)
              elif self.round_steps_count2 <= -1:
                  self.move_anti_clockwise_2()
                  self.round_steps_count2 += 1
                  print(self.round_steps_count1, self.round_steps_count2)
```

Slika 4.12. *Python* kod metode pomoću koje se preko izmjerenog broja koraka pomiču koračni motori

U metodi *move_motors_by_step_count(self)* (slika 4.12.) definira se *while* petlja koja se izvršava, dok su varijable instance *self.round_steps_1 i self.round_steps_2* različite od

nule. Ako je broj koraka pozitivan, koračni motor se pomiče u smjeru kazaljke na satu te se brojač koraka umanji za jedan. Kada je broj koraka negativan broj, koračni motor se pomiče u obrnutom smjeru od kazaljke na satu te se na brojač koraka povećava za jedan. Kada varijable instance postignu vrijednost nule, petlja se prekida i položaj motora se tada nalazi u traženoj x-y točki u koordinatnom sustavu. Kod je napisan tako da se pojedini koračni motori mogu zajedno gibati te individualno ovisno o izračunatom broju koraka koji motori moraju napraviti da zauzmu željeni kut pri kojem se kemijska olovka nalazi u traženom položaju.

4.3.1. Testiranje implementacije matematičkog modela

Nakon definiranih metoda za pokretanje motora i metode u koju je implementiran matematički model provodi se eksperimentalno testiranje rada manipulatora. Kako bi se testiranje provelo, potrebno je napraviti još jednu metodu unutar razreda *Plotter.* Metoda se zove *drive_xy(self)* (slika 4.13.) te se pomoću nje, korištenjem tipkovnice, mehanizam manipulatora može kretati u x-y smjeru koordinatnog sustava u koracima od 1 milimetra.

```
class Plotter():
        ---odrezano---
      def drive_xy(self):
          while True:
              key = readchar.readchar()
              if key == "q":
                  return
              elif key == "w":
                  self.current_y = self.current_y + 0.1
              elif key == "s":
                  self.current_y = self.current_y - 0.1
              elif key == "d":
                  self.current_x = self.current_x + 0.1
              elif key == "a":
                  self.current_x = self.current_x - 0.1
              print('X_Y: ', round(self.current_x), round(self.current_y))
              self.get_position_xy()
```

Slika 4.13. *Python* kod metode za pokretanje mehanizma manipulatora u koordinatnom sustavu manipulatora s tipkovnicom računala

Ako želimo da se kemijska olovka pomakne od trenutnog položaja u pozitivnom smjeru y potrebno je stinuti tipku ‘w’, u negativnom smjeru osi y tipku ‘s’ , u pozitivnom smjeru osi x tipku ‘d’ , te u negativnom smjeru osi x tipku ‘a’. Upravljanje se događa tako da se od početnog položaja dodaje vrijednost od jednog milimetra te taj položaj postaje trenutni. Sama kretnja se odvija u metodi *get_position_xy()* u kojoj je implementiran

matematički model manipulator te definiran način na koji se koračni motori pokreću kako bi zauzeli željeni položaj.

Kako bi se testirale kretnje manipulatora potrebno je napraviti glavnu skriptu u koju se unosi razred *Plotter.* Unutar glavne skripte (slika 4.14.) za pokretanje manipulatora stvara se novi objekt razreda *Plotter.* Nakon toga se pozivaju metode unutar objekta. Prva korištena metoda je *move_to_start_point()* s kojom se sami mehanizam postavlja u početni položaj, a druga metoda je *drive_xy()* s kojom se pokreće manipulator u x-y smjeru koordinatnog sustava.

```
from Plotter import *

plotter = Plotter()
plotter.move_to_start_point()
plotter.drive_xy()
```

Slika 4.14. *Python* kod glavne skripte za pokretanje manipulatora

Napokon, potrebno je sve napisane skripte učitati na *Raspberry Pi.* Povezivanjem računala s *Raspberry Pi-om* preko *SHH* mrežnog protokola omogućeno je slanje datoteka preko *SCP (Secure Copy Protocol)* protokola [19]. Samo slanje datoteka je jednostavno te se obavlja preko prozora naredbenog retka (*Command Prompt)* računala s kojim je povezan *Raspberry Pi,* naredbom:

scp \put_do_datoteke\ime_skripte_sa_razredom pi@ime_ili_IP_raspberryPia:

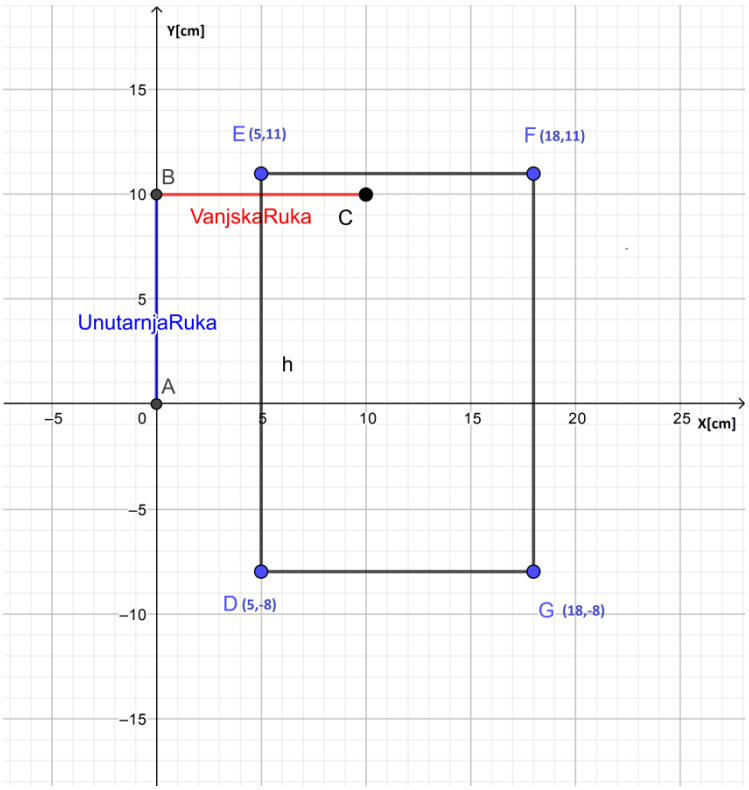
scp \put_do_datoteke\ime_glavene_skripe pi@ime_ili_IP_raspberryPia:

Nakon što se skripte prebace na *Raspberry Pi* u prozoru za upravljanje *Raspberry Pi-om* je potrebno upisati naredbu:

python3 naziv_glavne_skripte.py

Ova naredba pokreće skriptu pomoću koje se upravlja mehanizmom manipulatora preko prozora za upravljanje *Raspberry Pi-om.* Pritiskanjem tipki unutar prozora za upravljanje *Raspberry Pi-om* se kemijska olovka pokreće u x-y smjerovima koordinatnog sustava te se ispisuju trenutni položaji kemijske olovke u x-y koordinatama i kutevi koračnih motora koje su motori zauzeli. U konačnici se može testirati jeli mehanizam pravilno radi, odnosno jesu li kretnje kemijske olovke pravilne u odnosu na ishodište koordinatnog sustava. Također se eksperimentalnim testiranjem može odrediti u kojem

području granica, manipulator može raditi s implementiranim matematičkim modelom. Kako je cilj manipulatora da ispisuje fotografije potrebno je odrediti pravokutnu površinu unutar koje manipulator pravilno raditi. Tako se eksperimentalnim testiranjem i pomicanjem manipulatora odredilo na kojem području manipulator najbolje radi (slika 4.15.). Same granice se očitavaju preko prozora za upravljanje *Raspberry Pi* gdje se ispisuju vrijednosti trenutnog položaja kemijske olovke.

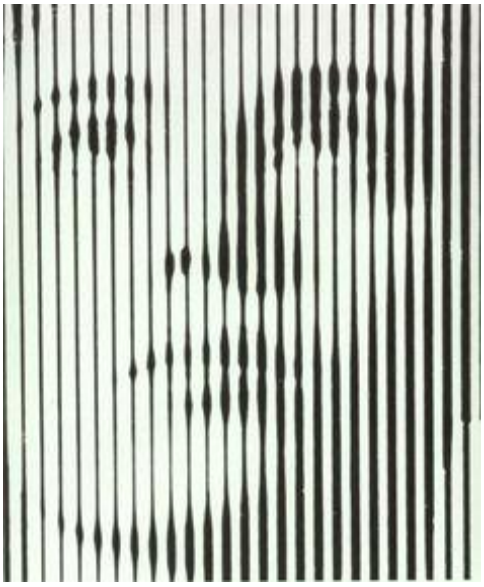


Slika 4.15. Područje rada manipulatora s implementiranim matematičkim modelom

Na slici 4.15. se vidi da matematički model najbolje radi u području prvog i četvrtog kvadranta koordinatnog sustava. Granice područja rada manipulatora određuju točke D, E, F i G. Stoga minimalan x, koji predstavlja lijevu granicu pravokutnika, iznosi 5 centimetara, maksimalan x iznosi 18 centimetara i on predstavlja desnu granicu, dok je minimalan y 8 centimetara i predstavlja donju granicu, a maksimalan je 11 centimetara te on predstavlja gornju granicu pravokutnika. Dolazi se do zaključka da su dimenzije u kojima je moguće ispisivati fotografije 13x19 centimetara što je malo manje od dimenzija A5 papira.

4.4. Programska podrška obrade fotografije

Programska podrška obrade fotografije je napisana u programskom jeziku *Python* s pripadajućim bibliotekama. Sama programska podrška se nalazi na računalu koji je povezan s *Raspberry Pi-om* te se ona vrši na njemu, a ne *Raspberry Pi-u* kako bi se ubrzala obrada fotografije. Ideja obrade fotografije je obraditi fotografiju sa slikarskom tehnikom sjenčanja *(eng. hatching)*(slika 4.16).



Slika 4.16. Slikarska tehnika sjenčanja[20]

Ovom slikarskom tehnikom se pomoću crtanja linija postiže tonalna razlika između svijetlih i tamnih područja. Što su linije bliže jedna drugoj postiže se tamniji ton. Na drugu stranu ako su linije udaljenije jedna od druge postiže se svjetliji ton. Sjenčanje se može postizati na različite načine. Postoji linearan način sjenčanja, ukršteno sjenčanje, te sjenčanje zakrivljenim linijama. Ideja manipulatora je da on ispisuje fotografiju linearnim sjenčanjem . Linearno sjenčanje je sjenčanje s paralelnim ravnim linijama koje mogu biti okomite, vodoravne ili pod nekim kutem [21]. Zbog ograničenosti konstrukcije i matematičkog modela manipulator ispisuje obrađene fotografije samo okomitim paralelnim linijama. Postoje dva tona sjenčanja koje manipulator ima mogućnost ispisivanja. Tonovi sjenčanja su određeni udaljenostima okomitih linija. Tako je ideja da manipulator crta linije tamniji tonova na način da linije budu udaljene jedna od druge pola milimetra, dok će svijetli tonovi biti paralelne linije koje su udaljene po jedan milimetar.

Fotografija se obrađuje na način da se prolaskom kroz stupce koordinatnog sustava fotografije odredi koje vrijednosti piksela će biti nacrtane svijetlim, odnosno tamnim tonovima. Nakon toga se x i y vrijednosti očitanih piksela grupiraju u liste koje predstavljaju točke u koordinatnom sustavu fotografije koje određuju jednu okomitu liniju ispisivanja. Koordinate piksela koji će se crtati je potrebno transformirati iz koordinatnog sustava fotografije u koordinatni sustav manipulatora i to u područje koje je prethodno određeno kao ono u kojem manipulator najbolje radi. Dobivene koordinate točaka će u konačnici predstavljati putanju gibanja manipulatora prilikom ispisa fotografije. Dodatno je potrebno dobiti koordinate točaka po kojim se manipulator giba kada dolazi u poziciju za ispis sljedeće linije. Sve putanje, odnosno koordinate točaka koje predstavljaju linije po kojima se manipulator giba je potrebno poslati na *Raspberry Pi*. Na kraju je potrebno napraviti novu metodu unutar razreda *Plotter* pomoću koje se dobivene točke zadaju manipulatoru te se u konačnici događa ispis fotografije.

4.4.1. *Python* skripta za obradu fotografije s pripadajućim *Python* bibliotekama

Nakon kratkog objašnjenja na čemu se temelji obrada fotografije potrebno je pojasniti kako je sama obrada fotografije napravljena korak po korak. Budući da se cijela obrada izvršava s *Python* programskim jezikom potrebno je napraviti *Python* skriptu u koju se prvo unose sve potrebne *Python* biblioteke. *Python* skripta je nazvana *Image_processing.py.* Biblioteke koje se koriste su:

- **matplotlib**

Ovom *Python* bibliotekom se omogućuje jednostavni ispis i vizualizaciju 2D grafova [22]. Biblioteka se koristi za prikazivanje rezultata obrade fotografije, gdje se pomoću funkcija iz biblioteke, crtaju točke i linije po kojima se manipulator mora gibati kako bi ispisao fotografiju.

- **subprocess**

Pomoću biblioteke *subprocess* omogućuju se stvaranje novih procesa unutar *Python* skripte [23]. Ovom bibliotekom u omogućuje se pristup naredbenom retku računala u kojem se unosi naredba kojom će se poslati rezultati obrade slike na *Raspberry Pi.*

- ***numpy***

Biblioteka *numpy* služi za korištenje kompleksnih matematičkih funkcija [24]. Koriste se funkcije za linearnu interpolaciju pomoću kojih se obrađuju koordinate točaka po kojima se manipulator giba.

- ***openCV***

OpenCV je *Python* biblioteka koja služi za računalnu obradu fotografije ili videa [25]. Sama biblioteka je otvorenog koda te je dostupna svima na korištenje. U ovom radu se ona koristi za prikaz fotografija koje se obrađuju.

- ***PIL(Python Image Library)***

PIL(Python Image Library) je biblioteka koja nudi funkcije za pristup i manipulaciju slikama [26]. Biblioteka se u ovom radu koristi za očitavanje vrijednosti piksela pomoću kojih će se definirati točke gibanja manipulatora.

Također se koriste i biblioteke *math* i *json* koje su objašnjene u poglavlja o programskoj podršci upravljanja manipulatora. U *Python* skripti za obradu fotografije se unose sve navedene biblioteke (slika 4.17).

```
from numpy import interp
import json
import numpy as np
import math
from PIL import Image
import cv2
import matplotlib.pyplot as plt
import subprocess
```

Slika 4.17. *Python* kod unosa potrebnih biblioteka za programske podršku obrade fotografije

4.4.2. *Python* razred za obradu fotografije

Kako bi postajala mogućnost obrade više fotografija istovremeno, koristi se metoda objektno orijentiranog programiranja. Unutar skripte za obradu fotografije nakon unosa svih potrebnih *Python* biblioteka definira se razred *Image_processing.* U razredu se definiraju pojedine metode pomoću kojih se omogućuje obrada fotografije. Ovaj razred inicijalizira se putanjom do datoteke koja se nalazi na računalu i koja sadrži fotografiju koju je potrebno obraditi. Oblik koji mora sadržavati ulaz u razred je:

photo = 'Put_do_datoteke\ime_datoteke.jpg/png'

Putanja se sprema u varijablu instance *self.photo* u konstruktoru razreda (slika 4.18.).

U nastavku su opisane sve metode definirane unutar razreda za obradu fotografije.

```
class Image_processing():
    def __init__(self,photo):
        self.photo = photo
```

Slika 4.18. *Python* kod razreda za programsku podršku obrade fotografije

4.4.2.1. Metoda za učitavanje fotografije u obradu

Prva metoda koja je definirana unutar razreda za obradu fotografije je i*mport_image(self)* (slika 4.19.).

```
class Image_processing():
    ---odrezano---
    def import_image(self):
        self.image = cv2.imread(self.photo,cv2.IMREAD_GRAYSCALE )
        (self.height, self.width) = self.image.shape[:2]
        # Rotiranje ako je širina veća od visine fotografije
        if self.width > self.height:
            self.image= rotate_bound(self.image, angle=-90)
            (self.height, self.width) = self.image.shape[:2]
            cv2.imshow('Imported Image',self.image)
            cv2.waitKey(0)
            cv2.destroyAllWindows()
        else:
            cv2.imshow('Imported Image', self.image)
            cv2.waitKey(0)
            cv2.destroyAllWindows()
        cv2.imwrite('gray.jpg', self.image)
```

Slika 4.19. *Python* kod metode za unos fotografije u obradu

Pomoću ove metode omogućuje se učitavanje slike, iz prethodno zadane datoteke u varijabli instance *self.photo* pomoću funkcije *cv2.imread(path ,flag)* [25]. S obzirom na to da manipulator koristi samo jednu boju, fotografija se učitava na način da se fotografija, ako je u boji, pretvori u crno-bijelu fotografiju. Razlog zbog kojeg je fotografiju potrebno pretvoriti u crno-bijelu boju je taj što su vrijednosti piksela fotografija u boji definirane u *RGB* vrijednosti prostoru boja što znači da je svaka boja predstavljena s tri komponente: crvena, zelena i plava [27]. Pretvaranjem fotografije u crno-bijelu boju pikseli su predstavljeni intenzitetom od 0 do 255, gdje vrijednost 0 predstavlja crnu boju, dok vrijednost od 255 predstavlja bijelu boju. Vrijednosti između predstavljaju nijanse sive boje.

Učitana fotografija se sprema u varijablu instance *self.image.* Nakon toga se unutar metode spremaju dimenzije fotografije u varijable instance *self.height* i *self.width* gdje

self.height predstavlja visinu fotografije, dok *self.width* predstavlja širinu fotografije. Sama visina i širina fotografije se određuje s metodom *shape()* koja se nalazi unutar biblioteke *OpenCV* [25]. Budući da je najpovoljnije područje rada manipulatora takvo da je dimenzija širine manja od dimenzije visine slike, fotografija se rotira za 90° ako joj je visina veća od širine. Za rotaciju fotografije koristi se funkcija *rotate_bounds(image, angle)* koja je u cijelosti prikazana u prilogu gdje se nalaze dodatne statične funkcije za obradu fotografije. U slučaju da se dogodi rotacija, ponovno se očitavaju dimenzije fotografije te se spremaju u već definirane varijable instance *self.height* i *self.widht.*
Na samom kraju se uz pomoć funkcije *cv2.imshow(window_name, image)* prikazuje fotografija kako bi se provjerilo je li fotografija uspješno učitana [25].

4.4.2.2. Metoda za automatizirano određivanje granica manipulatora

Kako manipulator ima točno određeno područje u kojem ispisuje, granice rada manipulatora koje smo odredili u prethodnim poglavljima potrebno je automatizirati u odnosu na veličinu fotografije koja se obrađuje. Stoga se definira nova metoda unutar razreda koja je nazvana *define_bounds(self)* (slika 4.20.).

```
class Image_processing():
     def __init__(self,photo):
       ---odrezano---
         #Granice pravokutnika unutar kojih manipulator ispisuje
     fotografije
         self.bounds = [5, -8, 18, 11]
     def define_bounds(self):
         # getting new bound (because of scaling)
         self.bounds[3] = round((self.bounds[1] + (self.bounds[2] -
     self.bounds[0]) * (self.height / self.width)), 1)
         if self.bounds[3] > 8:
             self.bounds[3] = 8
             self.bounds[2] = round(self.bounds[0] + (self.bounds[3] -
     self.bounds[1]) * self.width / self.height)
         self.bounds[3] += 1.3
         if (self.height / self.width) * 10 > 14:
             print('Please import photo where ratio of height and
     width of image is not bigger then 1.4')
```

Slika 4.20. *Python* kod metode za automatizirano određivanje granica rada manipulatora

U konstruktoru se u varijablu instance *self.bounds* spremaju granice područja u kojem manipulator najbolje radi. Prvi član liste predstavlja minimalan mogući x , drugi član minimalan mogući y, treći član maksimalan mogući x, dok četvrti član predstavlja najveći

mogući y. Širina pravokutnog područja koja definira područje ispisa fotografije iznosi 13 centimetara, dok visina iznosi 19 centimetara te je maksimalan mogući omjer visine i širine fotografije oko 1.4. Visina pravokutnog područja se skalira s omjerom visine i širine fotografije.. Ako je kojim slučajem omjer veći od navedenog, ispisuje se poruka koja ukazuje da se trebaju promijeniti dimenzije učitane fotografije jer obrada slike neće biti najbolja, ali će se svejedno provesti definiranje granica i daljnja obrada. Na kraju se ispisuju vrijednost novih granica manipulatora.

4.4.2.3. Metoda za crtanje gornjeg okvira na fotografiji

Ovom metodom se na učitanoj fotografiji crtaju dva gornja okvira. Eksperimentalnim testiranjem manipulatora se došlo do zaključka da se zbog nedostataka konstrukcije manipulatora, matematičkog modela i zbog loših karakteristika koračnih motora, prva linija koju manipulator ispisuje, ne ispisuje pravilno. Crtanjem linija na gornjem okviru fotografije se zapravo utječe na putanju rada manipulatora. Postiglo se to da manipulator crta prvo okvir te se na taj način izbjegava da se bitnije linije fotografije ne ispisuju nepravilno, odnosno okvir preuzima grešku ispisa. Metoda za crtanje gornjeg okvira na fotografiji je *upper_frame(self)* (slika 4.21.).

```
class Image_processing():
       ---odrezano---
      def upper_frame(self):
          threshold_line_dark = 0
          threshold_line_light = self.threshold_light - 10
          thickness_line_dark = int(self.height/65)
          thickness_line_light = thickness_line_dark*2
          start_point1 = (0, 0)
          end_point1 = (self.width, 0)
          color1 = (threshold_line_dark,threshold_line_dark ,
      threshold_line_dark)
          color2 = (threshold_line_light, threshold_line_light,
      threshold_line_light)
          start_point2 = (0, int((thickness_line_dark +
      thickness_line_light) / 2))
          end_point2 = (self.width, int((thickness_line_dark +
      thickness_line_light) / 2))
          self.line_image = cv2.line(self.image, start_point1,
      end_point1, color1, thickness_line_light)
          self.line_image = cv2.line(self.line_image, start_point2,
      end_point2, color2, thickness_line_dark)
          self.framed_photo = self.photo[:-4] + '_line.jpg'
          cv2.imwrite(self.framed_photo, self.line_image)
          cv2.imshow('line_image', self.line_image)
          cv2.waitKey(0)
          cv2.destroyAllWindows()
```

Slika 4.21. *Python* kod metode za crtanje gornjeg okvira na fotografiji

Koristi se funkcija *cv2.line(image, start_point, end_point, color, thickness)* pomoću koje se crtaju okviri na fotografiji [25]. Prvi ulaz u funkciju je slika na koju se crta okvir, dok su drugi i treći ulaz su početna i krajnja točka linije. Četvrti ulaz definira boju linije. Sama boja je određena pragovima tonova koji su opisani u sljedećem poglavlju. Posljednji ulaz predstavlja širinu linije koja se računa na temelju visine fotografije kako bi linija bila jednaka na svakoj fotografiji. Na kraju se poziva funkcija *cv2.imshow(window_name, image)* kako bi se provjerilo je li okvir dobro nacrtan na fotografiji.

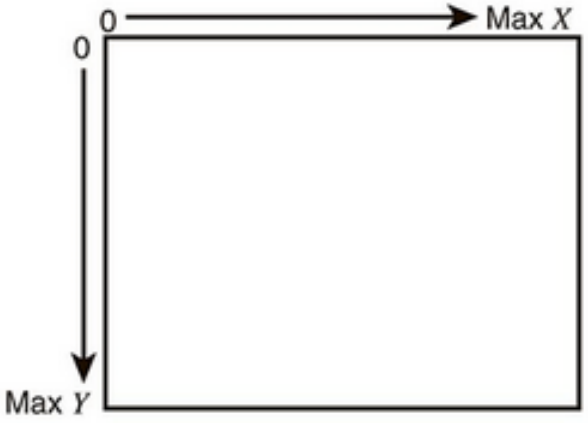
4.4.2.4. Metoda za određivanje koordinata točaka okomitih linija ispisivanja pomoću vrijednosti inenziteta piksela

Kako bi se dobile okomite linije kojima se manipulator giba kada ispisuje fotografiju stvara se metoda pomoću koje se vrši iteracija kroz piksele slike te se koordinata piksela spremaju u liste. Metoda za određivanje okomitih linija ispisivanja pomoću vrijednosti intenziteta piksela nazvana je *getting_points_pixels(self)* (slika 4.22.).

```
class Image_processing():
    def __init__(self,photo):
        ---odrezano---
        # Pragovi tonovoa za koje manipulator ispisuje fotografiju
        self.threshold_dark = 75
        self.threshold_light = 110
        # Prazna lista u koju se spemaju koordinate piksela
        self.Vertical_light_lines = []
        self.Vertical_dark_lines = []
    def getting_points_pixels(self):
        self.pixel_image = Image.open(self.framed_photo)
        self.pixel_image = self.pixel_image.convert("L")
        self.pixels = self.pixel_image.load()
        for x0 in range(self.width):
            line = []
            line_light = []
            for y0 in range(self.height):
                if self.pixels[x0, y0] <= self.threshold_dark:
                    line.append([x0, y0])
                if self.pixels[x0, y0] > self.threshold_dark and
self.pixels[x0, y0] <= self.threshold_light:
                    line_light.append([x0, y0])
            self.Vertical_dark_lines.append(line)
            self.Vertical_light_lines.append(line_light)
```

Slika 4.22. *Python* kod metode za određivanje koordinata točaka okomitih linija ispisivanja pomoću vrijednosti intenziteta piksela

Za očitavanje vrijednosti intenziteta piksela koristi se *PIL(Python Image Library)* biblioteka. U varijablu instance *self.pixel_image* učitava se fotografija metodom *open(image)* iz razreda *Image.* Ulaz u metodu je varijabla instance *self.framed_photo* koja predstavlja putanju do fotografije s nacrtanim gornjim okvirom iz prethodnog koraka. Nakon učitavanja fotografije stvara se varijabla instance *self.pixel* u koju se učitavaju vrijednosti piksela korištenjem metode *load()* iz razreda *Image.* Vrijednostima piksela se može pristupiti putem x-y koordinata piksela u koordinatnom sustavu fotografije. Koordinatni sustav fotografije je definiran kako je prikazano na slici 4.23.. Na slici se može vidjeti da je u gornjem lijevom kutu fotografije ishodište koordinatnog sustava.



Slika 4.23. Koordinatni sustav fotografije

Primjer pristupa pikselu u proizvoljnoj x-y točki:

vrijednost_piksela = self.pixel[x,y]

Budući da manipulator radi s dva tona boja potrebno je piksele fotografije podijeliti na one koje ne crtamo, one koje crtamo u svijetlom tonu te one koje crtamo u tamnom tonu. Stoga se u konstruktoru definiraju dva praga intenziteta boje: *self.threshold_dark* i *self.thershold_light*. Ako je vrijednost piskela manja od vrijednosti *self.threshold_dark* piksel treba biti nacrtan u tamnom tonu. S druge strane, ako je vrijednost piksela veća od vrijednosti *self.threshold_dark,* a manja od vrijednosti *self.threshold_light,* piksel se crta u svijetlom tonu. Piksele čija je vrijednost veća od *self.threshold_light* se preskaču. Pragovi ovise o fotografiji koja se obrađuje pa ih korisnik mora sam procijeniti prije obrade fotografije. Kako smo se odlučili obraditi fotografiju samo s okomitim linijama, pikseli se grupiraju po stupcima koordinatnog sustava slike. Tako se kroz iteraciju stupcem slike u varijablu *line* spremaju koordinate tamnih piksela, a u varijablu *line_light* koordinate svijetlih piskela. Dobivene liste s koordinatama piksela predstavljaju putanju jedne okomite linije ispisivanja. Nakon obrade pojedinog stupca fotografije, liste tamnih piksela se spremaju u varijablu instance *self.Vertical_dark_lines*, dok se liste svijetlih piksela spremaju u varijablu *self.Vertical_light_lines*.

4.4.2.5. Metode za transformaciju koordinata točaka okomitih linija ispisivanja iz koordinatnog sustava fotografije u koordinatni sustav manipulatora

Budući da se dobivene okomite linije ispisivanja nalaze u koordinatnom sustavu fotografije potrebno ih je pretvoriti u koordinatni sustav manipulatora. Stoga je

napravljena nova metoda unutar razreda koja je nazvana *image_transformation(self)* (slika 4.24.).

```
class Image_processing():
    def __init__(self,photo):
      ---odrezano---
        #Granice pravokutnika unutar kojih manipulator ispisuje
    fotografije
        self.bounds = [5, -8, 18, 11]
        # Prazne liste u koje se spremaju transformirane točke
        self.Vertical_light_lines_transformed = []
        self.Vertical_dark_lines_transformed = []
    def image_transformation(self):
        for line in self.Vertical_dark_lines:
            xy = []
            for points in line:
                x = interp(points[0], [0, self.width],
    [self.bounds[0], self.bounds[2]])
                y = interp(points[1], [0, self.height],
    [self.bounds[3], self.bounds[1]])
                x_y = [x, y]
                xy.append(x_y)
            self.Vertical_dark_lines_transformed.append(xy)
        for line in self.Vertical_light_lines:
            xy = []
            for points in line:
                x = interp(points[0], [0, self.width],
    [self.bounds[0], self.bounds[2]])
                y = interp(points[1], [0, self.height],
    [self.bounds[3], self.bounds[1]])
                x_y = [x, y]
                xy.append(x_y)
            self.Vertical_light_lines_transformed.append(xy)
```

Slika 4.24. *Python* kod metode za transformaciju koordinata točaka okomitih linija ispisivanja iz koordinatnog sustava fotografije u koordinatni sustav manipulatora

Transformacija se vrši funkcijom za linearnu interpolaciju *interp(x, xp, fp)* koja se nalazi u *Python* biblioteci *numpy* [24]. Interpolacijom se koordinate piksela iz koordinatnog sustava fotografije, mapiraju u koordinatni sustav manipulatora unutar prethodno određenih granica područja ispisivanja fotografije koje su definirane varijablom instance *self.bounds*. U varijablu instance s*elf.Vertical_dark_lines_transformed* se spremaju nove koordinate piskela koje se nalaze u koordinatnom sustavu manipulatora i predstavljaju točke po kojima se manipulator giba kada ispisuje tamne tonove, dok se u varijablu instance *self.Vertical_light_lines_transformed* spremaju točke koje predstavljaju točke kojima se manipulator giba kada ispisuje svijetle tonove. fotografije.

Kako se transformacijom iz koordinatnog sustav fotografije u koordinatni sustav manipulatora dobiju vrijednosti koordinata točaka u nezgodnim decimalnim brojevima, potrebno ih je zaokružiti na određene vrijednost. Stoga se stvara nova metoda *round_drawing_points(self)* (slika 4.25.).

```
class Image_processing():
    def __init__(self,photo):
         ---odrezano---
        # Vrijednosti na koje ukazuju na kojoj su udaljenosti paralelene linije
ispisivanja
        self.width_between_line_light = 0.1   # za svijetle tonove 0.1 cm
        self.width_between_line_dark = 0.05  # za tamne tonove 0.05 cm
        self.width_y = 0.05
       # Liste u koje se spremaju zaokružene točke u koordinatnom sustavu
       manipulatora
       self.Vertical_dark_lines_round = []
       self.Vertical_light_lines_round = []

       def round_drawing_points(self):
           for line in self.Vertical_dark_lines_transformed:
               round_line = []
               for x_y in line:
                   xy = []
                   x = round_nearest(x_y[0], self.width_between_line_dark)
                   y = round_nearest(x_y[1], self.width_y)
                   xy.append(x)
                   xy.append(y)
                   round_line.append(xy)
               self.Vertical_dark_lines_round.append(round_line)
           for line in self.Vertical_light_lines_transformed:
               round_line = []
               for x_y in line:
                   xy = []
                   x = round_nearest(x_y[0], self.width_between_line_light)
                   y = round_nearest(x_y[1], self.width_y)
                   xy.append(x)
                   xy.append(y)
                   round_line.append(xy)
               self.Vertical_light_lines_round.append(round_line)
```

Slika 4.25. *Python* kod metode za zaokruživanja koordinata točaka okomitih linija na željene vrijednosti

Varijablama instanci *self.width_between_line_light* i *self.width_between_dark_line* se definira kako zaokružujemo x koordinatu točaka. Ovim varijablama se zapravo određuje udaljenost između okomitih linija. Za tamne tonove je definirano da udaljenost između linija iznosi 0.05 centimetara, dok za svijetle tonove iznosi 0.1 centimetar. Tako se x koordinata točke zaokružuje na najbliži višekratnik 0.05 za svijetle tonove te na 0.1 za tamne tonove. Također se unutar konstruktora definira i varijabla instance *self.width_y* kojom se definira udaljenost točaka u smjeru osi y. Kako bi manipulator prilikom ispisivanja fotografija radio što manju grešku, udaljenost se postavlja na 0.05 centimetara. Zaokruživanje se vrši pomoću statičke funkcije *round_nearest(x , a)* gdje prvi ulaz predstavlja x ili y vrijednost točke, dok drugi ulaz predstavlja decimalan broj na

čiji višekratnik želimo zaokružiti vrijednosti točke. Funkcija je prikazana u prilogu. Linije sa zaokruženim točkama se spremaju u varijablu instance *self.Vertical_dark_lines_round* za linije koje predstavljaju tamne tonove, dok se okomite linije koje predstavljaju svijetle tonove spremaju u varijablu instance *self.Vertical_light_lines_round.*

Nakon zaokruživanja točaka, lista okomitih linija može sadržavati više jednakih točaka, odnosno može postojati više jednakih okomitih linija. Njih je potrebno ukloniti jer usporavaju rad manipulatora pa je zbog toga napravljena metoda *drawing_points_processing(self)* (slika 4.26.).

```
class Image_processing():
    def __init__(self,photo):
      # Brojka s kojom dobivamo cijeli broj zaokruženih točaka gibanja manipulatora
      self.get_round_01 = 10
      self.get_round_005 = 100

      def drawing_points_processing(self):

          Removed_same_lines_in_column_dark =
      remove_same_lines_x(self.Vertical_dark_lines_round, self.get_round_005)
          Removed_same_lines_in_column_light =
      remove_same_lines_x(self.Vertical_light_lines_round,self.get_round_01 )
          Splitting_lines_from_one_row_dark =
      split_lines(Removed_same_lines_in_column_dark)
          Splitting_lines_from_one_row_light =
      split_lines(Removed_same_lines_in_column_light)
          Removed_small_lines_dark =
      remove_small_lines(Splitting_lines_from_one_row_dark)
          Removed_small_lines_light=
      remove_small_lines(Splitting_lines_from_one_row_light)
          Removed_same_points_in_one_line_dark =
      remove_same_points_in_line(Removed_small_lines_dark,
      self.get_round_005,
      self.get_round_005,
      self.width_between_line_dark,
      self.width_y)
          Removed_same_points_in_one_line_light =
      remove_same_points_in_line(Removed_small_lines_light,
      self.get_round_01,
      self.get_round_005
      self.width_between_line_light,
      self.width_y)
          self.Final_lines_dark =
      remove_small_lines(Removed_same_points_in_one_line_dark)
          self.Final_lines_light
      =remove_small_lines(Removed_same_points_in_one_line_light)
```

Slika 4.26. *Python* kod metode za uklanjanje viška točaka i linija koje usporavaju rad manipulatora

Unutar metode se redom pozivaju razne statičke funkcije koje su prikazane u cijelosti u prilogu. Prva statička funkcija koja se poziva je *remove_same_lines_x(list, get_round_number_of_x).* Ovom funkcijom se uklanjaju sve liste okomitih linije koje sadržavaju iste koordinate točaka. Iduća statička funkcija koja se poziva je *split_lines(list)*. Pomoću ove funkcije se provjerava postoje li dvije uzastopne točke udaljene vise od 0.05 centimetara u smjeru y osi. Ako postoje takve točke, linija se dijeli na dvije nove linije sve dok niti jedna linija nema uzastopnih točaka udaljenijih od 0.05

centimetar. Nakon toga se poziva statička *remove_small_lines(list).* Pomoću ove funkcije se uklanjaju liste koje sadrže samo jednu točku budući da je za definiranje jedne linije ispisivanja potrebno imati minimalno dvije točke. Iduća statička funkcija koja se poziva je *remove_same_points_in_line(lines,get_round_number_of_x,get_round_number_of_y,number_of_decimal_x,number_of_decimal_y).* Pomoću ove funkcije se uklanjaju točke istih koordinata unutar jedne okomite linije ispisivanja. Na kraju se ponovno poziva statička funkcija remove*_small_lines(list)* zbog toga što se pozivanjem prethodnih funkcija može dogoditi da ponovno nastanu liste linija sa samo jednom točkom. U konačnici se rezultati, odnosno konačne liste okomitih linija ispisivanja spremaju u varijable instanci *self.Final_lines_dark i self.Final_lines_light*.

4.4.2.6. Metode za određivanje putanje gibanja manipulatora do početnog položaja sljedeće linije ispisivanja

Metoda s kojom se dobiva putanja gibanja manipulatora do prve točke ispisivanja okomitih linija nazvana je *get_move_to_new_drawing_line(self)* (slika 4.27.).

```
class Image_processing():
      def __init__(self,photo):
           ---odrezano---
          # Početna pozicija kemijske olovke
          self.default_position = [10.5, 9.2]
          # Definiranje praznih lista u koje se spremaju rezultati putanje gibanja
      manipulatora do nove linije ispisivanja
           self.Move_to_line_dark = []
          self.Move_to_line_light = []
       def get_move_to_new_drawing_line(self):

           x = self.default_position[0]
           y = self.default_position[1]
          for line in self.Final_lines_dark:
               first_point = line[0]
               to_new_line = list(zip(np.linspace(x, first_point[0], 100),
                              np.linspace(y, first_point[1], 100)))
               self.Move_to_line_dark.append(to_new_line)
               for points in line:
                   x= points[0]
                   y= points[1]

           x = self.default_position[0]
           y = self.default_position[1]
          for line in self.Final_lines_light:
              first_point = line[0]
              to_new_line = list(zip(np.linspace(x, first_point[0], 100),
                              np.linspace(y, first_point[1], 100)))
              self.Move_to_line_light.append(to_new_line)
              for points in line:
                  x = points[0]
                  y = points[1]
```

Slika 4.27. *Python* kod metode za dobivanje putanje gibanja manipulatora do početnog položaja sljedeće linije ispisivanja

Unutar konstruktora razreda se definira varijabla instance *self.default_position* koja predstavlja početni položaj kemijske olovke u koordinatnom sustavu manipulatora. Budući da se manipulator prije početka ispisivanja nalazi u početnim položaju, prvo je potrebno dobiti putanju gibanja manipulatora od početnog položaja do prve točke okomite linije ispisivanja. Nakon toga se dobivaju putanje gibanja između posljednje točke ispisane okomite linije do prve točke sljedeće linije ispisivanja. Kako bi se dobile putanje koristi se funkcije *linespace(start, stop, num=100)* iz *Python* biblioteke *numpy.* Pomoću ove funkcije se dobiva određeni broj točaka između okomitih linija ispisivanja po kojima se manipulator giba kada ne ispisuje fotografiju. Rezultati dobivenih putanja gibanja manipulatora između ispisivanja okomitih linija se spremaju u varijable instance *self.Move_to_line_dark* za okomite linije koje predstavljaju tamne tonove fotografije, dok se varijablu instance *self.Move_to_line_light* spremaju za svijetle tonove. Kako se prethodnom metodom dobivaju nezgodni decimalni brojevi potrebno je zaokružiti vrijednosti točaka i u konačnici ukloniti višak točaka po kojima se giba manipulator do nove linije ispisivanja, analogno metodama u odjeljku 4.4.2.5.. Stoga je stvorena metoda *move_to_line_processing(self)* (slika 4.28.).

```
class Image_processing():
    def __init__(self,photo):
        ---odrezano---
      def move_to_line_processing(self):

          Round_move_to_new_line_dark = round_points(self.Move_to_line_dark,
      self.width_between_line_dark,self.width_y)
          Round_move_to_new_line_light = round_points(self.Move_to_line_light,
      self.width_between_line_light, self.width_y)
          self.Final_move_to_new_line_dark =
      remove_same_points_in_line(Round_move_to_new_line_dark,
      self.get_round_005,
      self.get_round_005,
      self.width_between_line_dark,
      self.width_y)
          self.Final_move_to_new_line_light =
      remove_same_points_in_line(Round_move_to_new_line_light,
      self.get_round_01,
      self.get_round_01,
      self.width_between_line_light,
      self.width_y)
```

Slika 4.28. *Python* kod metode za obradu putanje gibanja manipulatora do početnog položaja sljedeće linije ispisivanja

4.4.2.7. Metode za prikaz rezultata obrade fotografije

Kako bi se provjerilo jesu li rezultati obrade fotografije pravilni, napravljene su dvije metode za prikaz rezultata. Prva metoda *plot_points(self)* (slika 4.29.) služi za prikaz dobivenih točaka po kojima se manipulator mora gibati kada ispisuje fotografiju. Rezultati su prikazani u koordinatnom sustavu manipulatora.

```
class Image_processing():
       ---odrezano---
      def plot_points(self):
          fig = plt.figure(figsize=(7,8))
          plt.xlabel('X [cm]')
          plt.ylabel('Y [cm]')
          plt.xscale("linear")
          plt.yscale("linear")
          for line in self.Final_lines_dark:
              for point in line:
                  plt.plot(point[0],point[1], marker='.', markersize=1,
      color="black")
          for line in self.Final_lines_light:
              for point in line:
                  plt.plot(point[0],point[1], marker = '.', markersize
      = 1, color = 'gray')
          # Draw manipulator arms
          # unutarnja ruka
          plt.plot([0, 0], [0, 10], linewidth=1, color='red')
          # vanjska ruka
          plt.plot([0, 10], [10, 10], linewidth=1, color='blue')
          plt.show()
          fig.savefig('Point_figure.png')
```

Slika 4.29. *Python* kod metode za prikaz rezultata točaka obrade fotografije

Unutar ove metode se pomoću funkcija iz biblioteke *matplotlib* prikazuju rezultati dobivenih točaka. Funkcija kojom se ispisuju točke na grafu koordinatnog sustava manipulatora je funkcija *plot(x, y, marker, markersize, color)* [22]. Točke kojima se manipulator treba gibati kada ispisuje linije tamnih tonova se ispisuju u crnoj boji, dok se točke kojima se manipulator giba kada ispisuje linje svijetlih tonova crtaju sivo. Unutar metode se crtaju i ruke manipulatora kako bi se dobio osjećaj veličine slike koju bi manipulator trebao ispisati u odnosu na konstrukciju manipulatora.

Iduća metoda je metoda *plot_drawing_lines(self)* (slika 4.30.) kojom se prikazuju okomite linije koje bi manipulator trebao ispisati.

```
class Image_processing():
       ---odrezano---
     def plot_drawing_lines(self):
         fig = plt.figure(figsize=(7,8))
         plt.xlabel('X [cm]')
         plt.ylabel('Y [cm] ')
         plt.xscale('linear')
         plt.yscale('linear')
         for line in self.Final_lines_dark:
             x_first = line[0][0]
             y_first = line[0][1]
             x_last = line[-1][0]
             y_last = line[-1][1]
             x = [x_first,x_last]
             y = [y_first,y_last]
             plt.plot(x,y,color= 'black', linewidth = 0.05)
         for line in self.Final_lines_light:
             x_first = line[0][0]
             y_first = line[0][1]
             x_last = line[-1][0]
             y_last = line[-1][1]
             x = [x_first, x_last]
             y = [y_first, y_last]
             plt.plot(x, y, color='black', linewidth = 0.05)
         # Draw manipulator arms
         # unutarnja ruka
         plt.plot([0, 0], [0, 10],linewidth = 1, color='red')
         # vanjska ruka
         plt.plot([0, 10], [10, 10], linewidth = 1, color='blue')
         plt.show()
         fig.savefig('Line_figure.png')
```

Slika 4.30. *Python* kod metode za prikaz okomitih linija obrađene fotografije

Unutar ove metode se pomoću funkcije *plot(x , y, color,linewidth)* crtaju linije kroz prvu i zadnju točku koje definiraju pojedinu liniju ispisivanja. Rezultati linija se crtaju u crnoj boji. Kako bi dobili bolji dojam onoga što bi manipulator trebao ispisivati debljine linija se postavljaju na 0.1 zato što kemijska olovka ima širinu ispisa od 0.1 centimetar.

4.4.2.8. Metoda za spremanje rezultata obrade fotografije

Budući da se obrada fotografije vrši na računalu koje je spojeno na *Raspberry Pi*, potrebno je napraviti metodu s kojom se spremaju rezultati obrade fotografije te se u konačnici šalju na *Raspberry Pi.* Metoda za spremanje rezultata je nazvana *Save_results(self)* (slika 4.31.). Unutar ove metode se stvara nova tekstualna datoteka pomoću *Python* naredbe *open()*. Prvi ulaz u metodu je ime datoteke s datotečnim nastavkom *.txt,* dok je drugi ulaz string *'w'* koji označava da se stvara nova datoteka za pisanje. Ime datoteke je postavljeno u varijabli instance *self.name_of_txt_file* tako da se na ime datoteke slike doda nastavak *_hatch.txt* kako bi za svaku obradu fotografije dobili novu datoteku s različitim imenom. Nakon toga se stvara rječnik koji je nazvan *points.*

Unutar rječnika se definiraju četiri liste u koje se spremaju rezultati. Tako se unutar liste *dark_points* spremaju liste okomitih linija koje predstavljaju tamne tonove fotografije, dok se unutar liste *move_to_new_line_dark* spremaju okomite linije koje predstavljaju svijetle tonove fotografije. Također su definirane liste *move_to_new_line_dark i move_to_new_line_light* koje sadržavaju putanje gibanja kojima manipulator dolazi do prve točke ispisivanja okomitih linija. U konačnici se poziva funkcija *json.dump()* iz biblioteke *json* s kojom se rječnik upisuje u datoteku koja je prethodno stvorena.

```
class Image_processing():
        ---odrezano---
    def Save_results(self):
        self.name_of_txt_file = self.photo[:-4] + '_hatch.txt'
        with open(self.name_of_txt_file, 'w') as point_dic:
            points = {}
            points["dark_points"] = self.Final_lines_dark
            points["move_to_new_line_dark"] =
self.Final_move_to_new_line_dark
            points["light_points"] = self.Final_lines_light
            points["move_to_new_line_light"] =
self.Final_move_to_new_line_light
            json.dump(points, point_dic, indent=8)
```

Slika 4.31. *Python* kod metode za spremanje rezultata obrade fotografije

4.4.2.9. Metoda za slanje rezultata obrade fotografije na *Raspberry Pi*

U konačnici je potrebna metoda s kojom se dobiveni rezultati spremljeni u tekstualnu datoteku šalju na *Raspberry Pi* . Metoda je nazvana *Send_results(self)* (slika 4.32.). Slanje rezultata je omogućeno s bibliotekom *subprocess* kojom se omogućuje pokretanje raznih procese na računalu. U prikazanoj metodi se pomoću funkcije *subprocess.run()* pokreće naredbeni redak *(Command Prompt)* računala u kojem se izvršava naredba kojom se preko *SCP (Secure Copy)* protokola datoteka s rezultatima obrade fotografije šalje na *Raspberry Pi* [23]. Kako bi se uspješno prebacili podatci na *Raspberry Pi* računalo prethodno mora biti povezano s *Raspberry Pi-om* preko *SSH* mrežnog protokola.

```
class Image_processing():
      ---odrezano---
    def Send_results(self):
        cmd = ['scp', 'C:\\Users\\Ivan\\PycharmProjects\\ploter\\'+
self.name_of_txt_file, 'pi@raspberrypi.local:']
        proc = subprocess.run(cmd, stderr=subprocess.PIPE,
stdout=subprocess.PIPE, shell=True)
```

Slika 4.32. *Python* kod metode za slanje rezultata obrade fotografije na Raspberry Pi

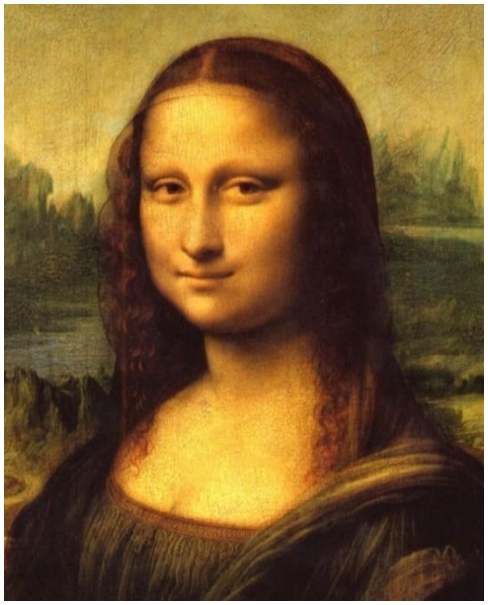
4.4.3. Obrada fotografije

U ovom poglavlju je prikazano kako obraditi fotografiju pomoću stvorenog razreda za obradu fotografije. Obrada je izvršena na dva primjera koja su naknadno i eksperimentalno ispisana s manipulatorom. Kako bi obradili slike potrebno je napraviti glavnu skriptu (slika 4.33.) u koju se unosi razred *Image_processing* iz skripte *Image_processing.py* te ostatak statičkih funkcija koje su definirane u njoj.

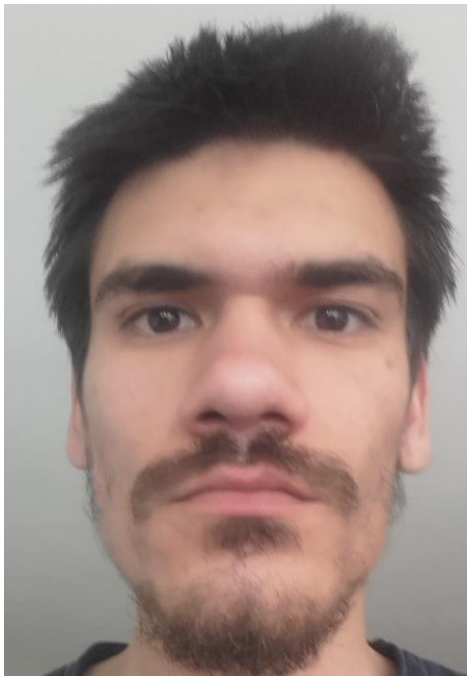
```
from Image_processing import *

image = 'mona.jpg'
mona_lisa = Image_processing(image)
mona_lisa.import_image()
mona_lisa.upper_frame()
mona_lisa.define_bounds()
mona_lisa.threshold_dark = 60
mona_lisa.threshold_light = 100
mona_lisa.getting_points_pixels()
mona_lisa.image_transformation()
mona_lisa.round_drawing_points()
mona_lisa.drawing_points_processing()
mona_lisa.get_move_to_new_drawing_line()
mona_lisa.move_to_line_processing()
mona_lisa.plot_points()
mona_lisa.plot_drawing_lines()
mona_lisa.Save_results()
mona_lisa.Send_results()

image = 'author.jpg'
author = Image_processing(image)
author.import_image()
author.upper_frame()
author.define_bounds()
author.threshold_dark = 90
author.threshold_light = 120
author.getting_points_pixels()
author.image_transformation()
author.round_drawing_points()
author.drawing_points_processing()
author.get_move_to_new_drawing_line()
author.move_to_line_processing()
author.plot_points()
author.plot_drawing_lines()
author.Save_results()
author.Send_results()
```

Slika 4.33. Python kod glavne skripte za obradu fotografije

Unutar koda glavne skripte za obradu fotografije se prvo obrađuje umjetničko djelo *Mona Lisa* (slika 4.34.). Prvo se definira ulaz u razred. Ulaz mora biti napisan kao tekst koji sadržava put do datoteke u kojoj se nalazi slika umjetničkog djela *Mona Lisa* te sami naziv datoteke. Stvara se novi objekt razredom *Image_processing(photo)* gdje je ulaz definiran varijablom *image*. Nakon toga se poziva metoda objekta *import_image()* pomoću koje se učitavaju zadane fotografije te se fotografija prikaže u crno-bijeloj boji. Analogno se na isti način, u drugom dijelu koda, obrađuje fotografija portreta autora (slika 4.35.).

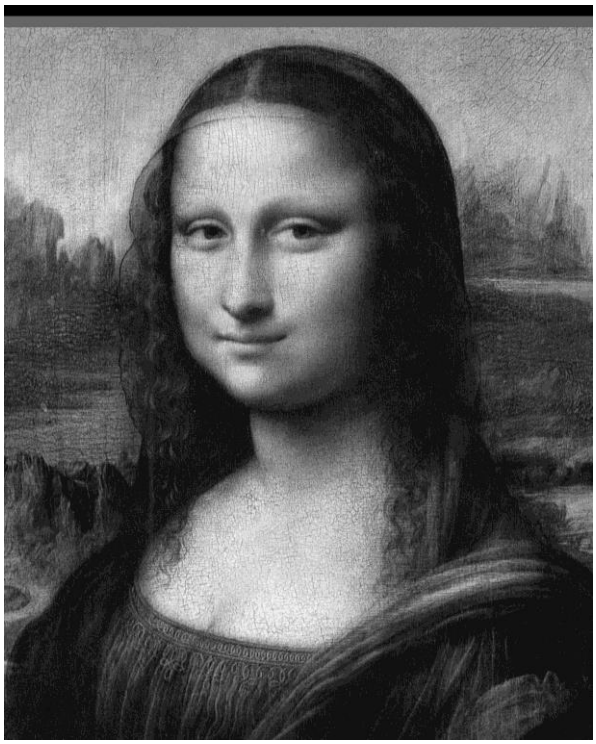


Slika 4.34. Umjetničko djelo *Mona Lisa* [28]

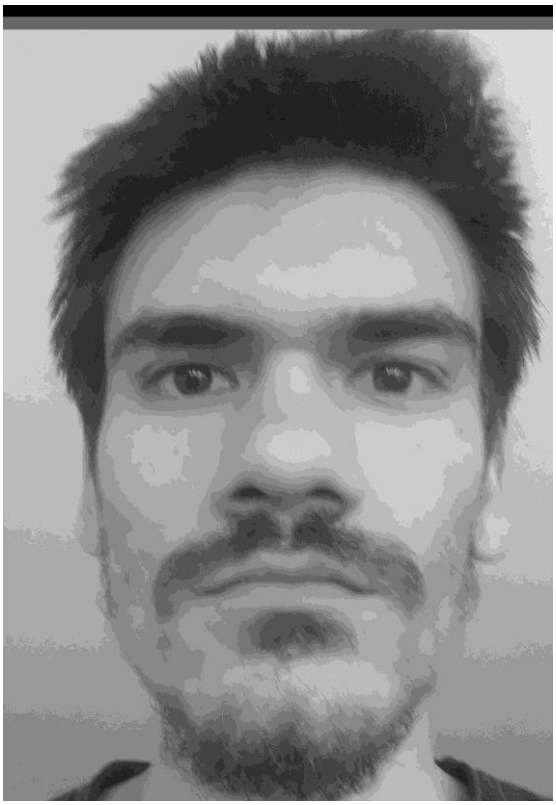


Slika 4.35. Fotografija autora diplomskog rada

Nakon unosa fotografije poziva se metoda *upper_frame()* koja na učitanim fotografijama crta gornji okvir na fotografiji te u konačnici prikaže rezultat fotografije u crno bijeloj boji s okvirom(slika 4.36., slika 4.37).



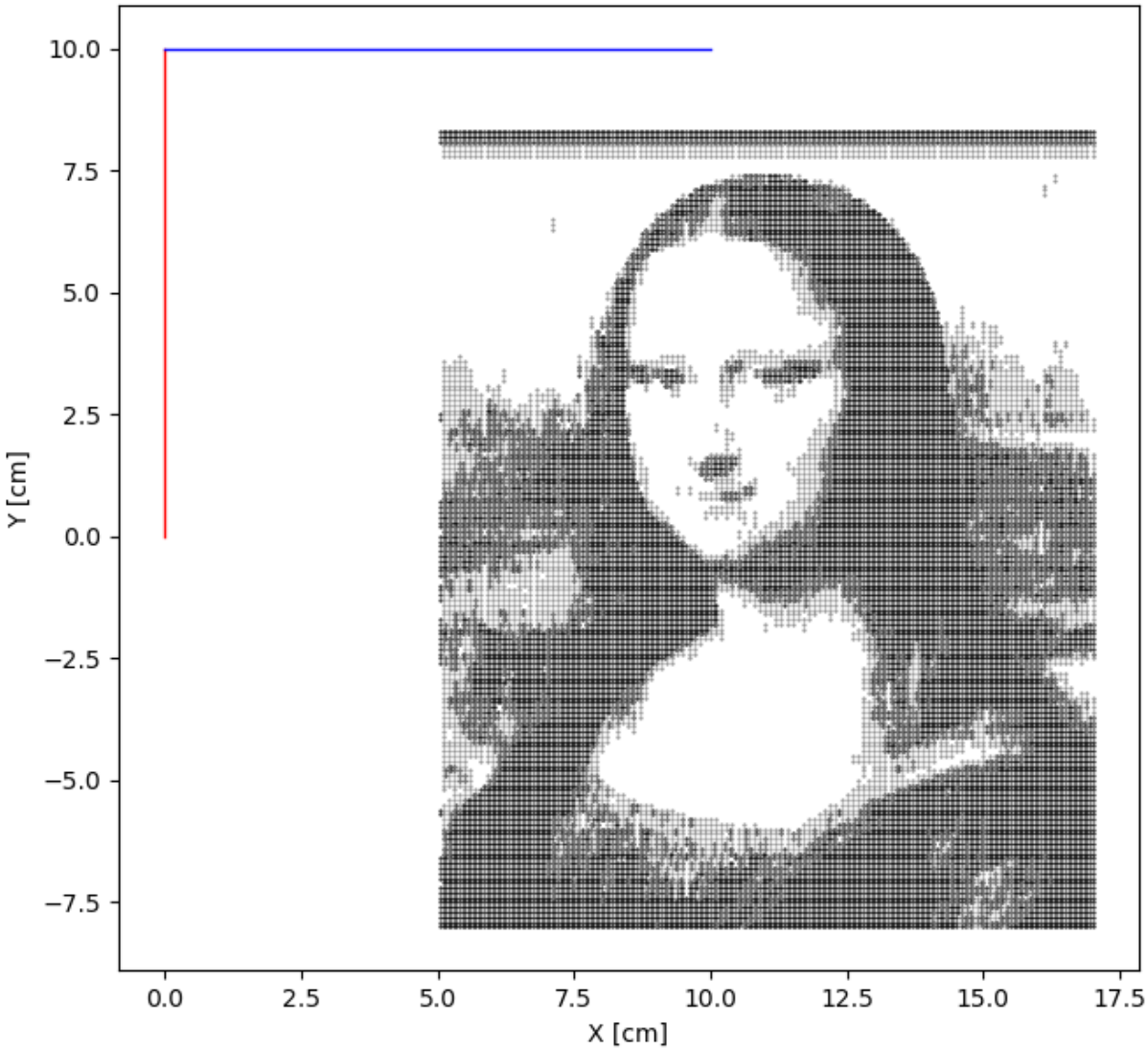
Slika 4.36. *Mona Lisa* s nacrtanim gornjim okvirom



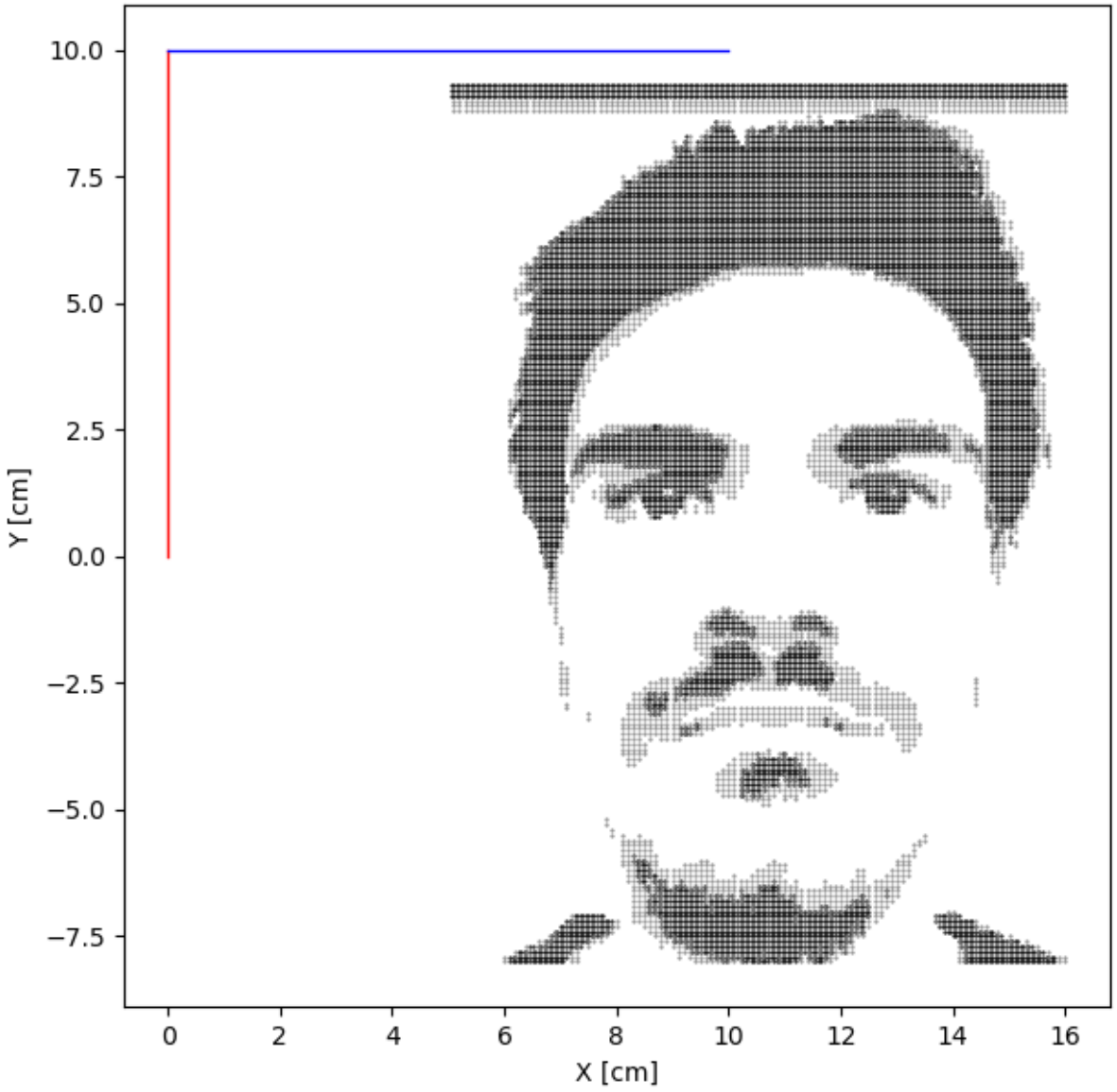
Slika 4.37. Fotografija autora diplomskog rada s nacrtanim gornjim okvirom

Nakon crtanja okvira poziva se metoda *define_bounds()* s kojom se definira područje ispisa fotografije u odnosu na veličinu fotografije. Kada se odredi područje dolazi se do glavnog dijela obrade fotografije gdje je potrebno definirati pragove s kojima se očitavaju pikseli na fotografiji. Pragovi se definiraju eksperimentalnim putem dok se ne dobiju zadovoljavajući rezultati koji se provjeravaju s metodama za prikaz rezultata obrade. Pragovi se definiraju mijenjanjem varijabli instance *self.threshold_dark i self.threshold_light* unutar razreda. Poželjno je za bolju obradu fotografije da fotografija ima što bolji kontrast između podloge i objekta fotografije koji se želi ispisati. Nakon definiranja pragova, poziva se metoda *getting_point_pixels()* koja iteracijom kroz piksele fotografije stvara listu točaka u koordinatnom sustavu fotografije koje definiraju koordinata točaka okomitih linija po kojima se manipulator giba kada ispisuje fotografiju. Kako je potrebno prebaciti točke iz koordinatnog sustava fotografije u koordinatni sustav manipulatora poziva se metoda *image_transformation().* Nakon što se prebace vrijednosti točaka iz jednog sustava u drugi, točke je potrebno zaokružiti na željene vrijednosti koje određuju udaljenost okomitih linija pa se poziva metoda *round_drawing_points().* Zaokruživanjem nastaje višak točaka i linija koje je potrebno ukloniti jer usporavaju rad manipulatora. Stoga se poziva metoda *drawing_points_processing().* Pozivanjem ove metode dobivaju se konačne koordinate točaka okomitih linija po kojima se manipulator giba kada ispisuje fotografiju. Nakon dobivanja okomitih linija potrebno je pozvati metodu *get_move_to_new_drawing_line().* S ovom metodom se dobivaju točke koje predstavljaju putanju gibanja do početnog položaja ispisivanja sljedeće linije ispisivanja. Same točke je potrebno obraditi zbog nastanka viška točaka koje usporavaju rad manipulatora pa se poziva metoda *move_to_line_processing()*.. Nakon dobivenih rezultata pozivaju se metode s kojima se prikazuju rezultati u koordinatnom sustavu manipulatora. Metode za prikaz rezultata koje se pozivaju su *plot_points(),* koja prikazuje točke po kojima se manipulator treba gibati da bi ispisao fotografiju (slika 4.38., slika 4.39.), i *plot_drawing_lines()* koja prikaziva okomite linije koje manipulator treba ispisati (slika 4.40., slika 4.41.) . Na rezultatima se može zaključiti da su slike dobro obrađene sa zadanim vrijednostima pragova. Na rezultatima se može vidjeti da su točke koje će se zadavati manipulatoru pravilno udaljene jedna od druge što ukazuje na dobru obradu fotografije. Vrijeme trajanja obrade fotografije ovisi o veličini fotografije. Što fotografija ima veću rezoluciju obrada traje duže, ali su rezultati tada bolji. U konačnici se poziva metoda *Save_results()*

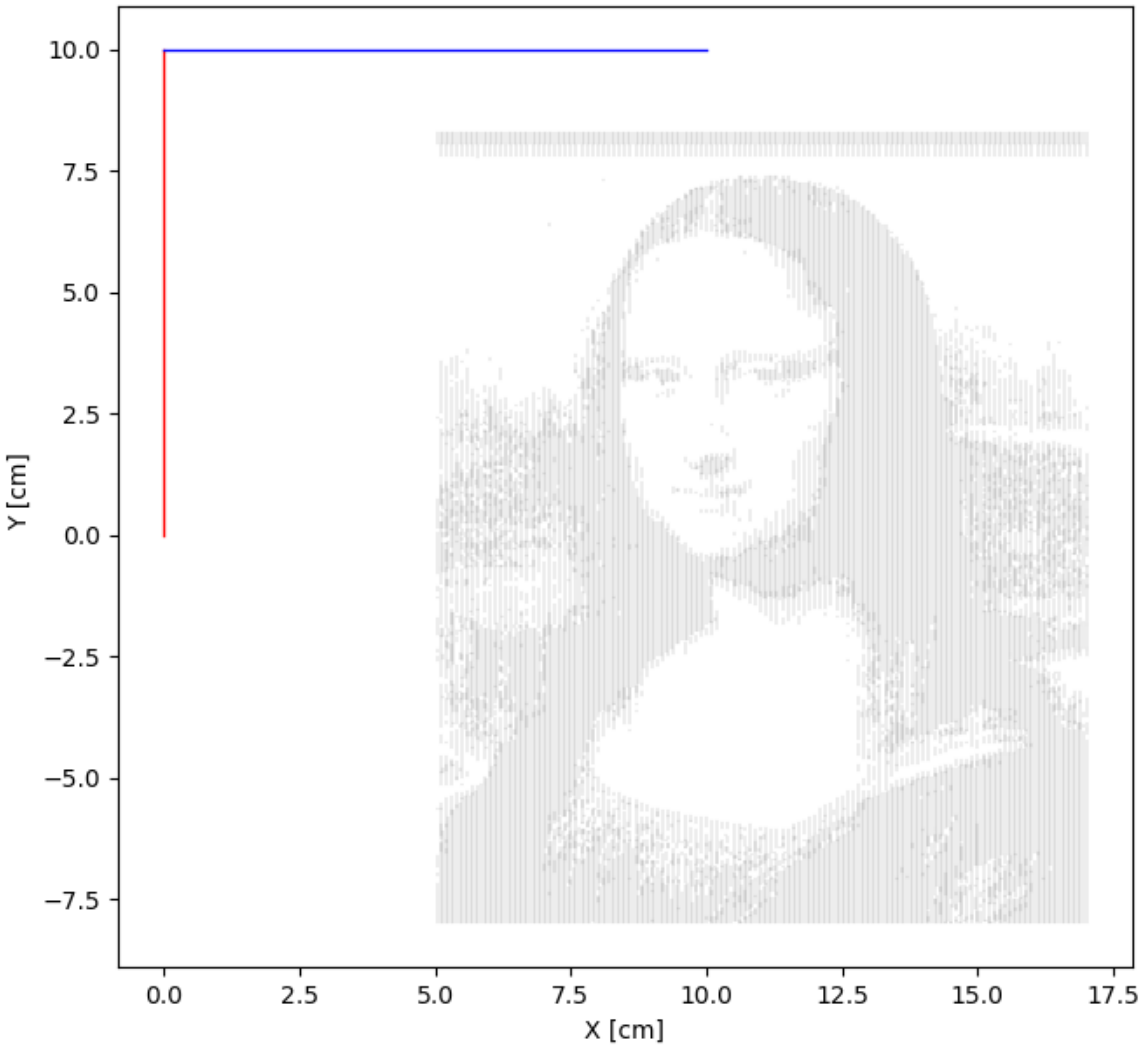
s kojom se spremaju rezultati na računalo te metoda *Send_results()* s kojom se pošalju spremljeni rezultati na povezani *Raspberry Pi.*



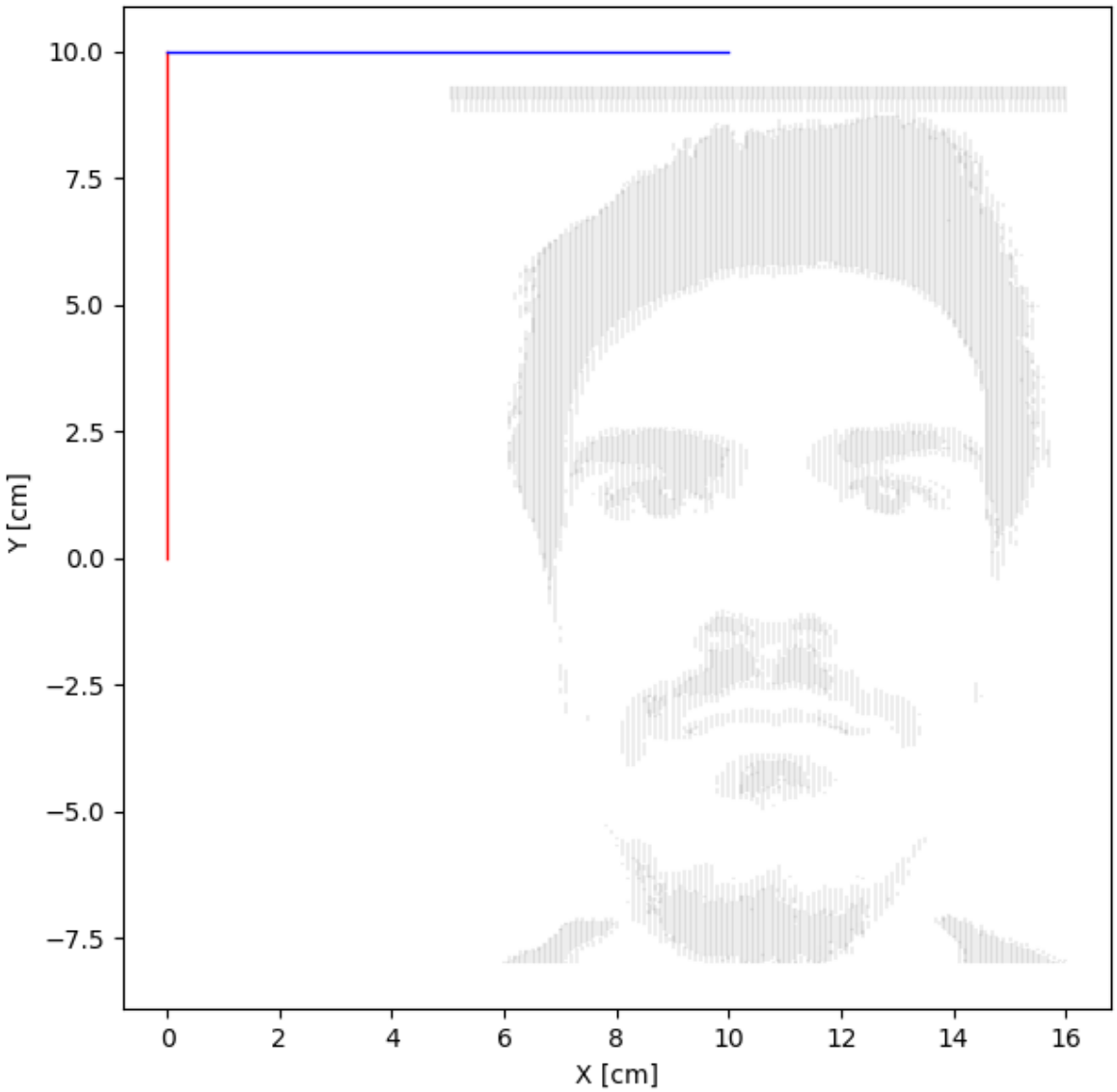
Slika 4.38. Prikaz točaka rezultata obrade umjetničkog djela *Mona Lisa*



Slika 4.39. Prikaz točaka rezultata obrade fotografije autora diplomskog rada



Slika 4.40. Prikaz rezultata linija obrade fotografije umjetničkog djela *Mona Lisa*



Slika 4.41. Prikaz rezultata linija obrade fotografije autora diplomskog rada

4.5. Implementacija rezultata obrade fotografije u programsku podršku upravljanja manipulatora

Nakon uspješne obrade fotografije, rezultate je potrebno implementirati u programsku podršku upravljanja manipulatora. Stoga se unutar razreda *Plotter* stvaraju nove metode koje obrađuju rezultate obrade fotografije te se manipulator prema rezultatima pokreće i vrši se ispis fotografije. Napravljene su dvije metode za ispisivanje linija. Prva metoda je nazvana *plot_dark_lines(self, file)* (slika 4.42.). Uloga ove metode je ispis okomitih linija koje predstavljaju tamne tonove fotografije. Ulaz u metodu je *file* koji treba sadržavati rezultate obrade fotografije koje su spremljene u datoteku u tekstualnom formatu. Sami ulaz mora sadržavati put do datoteke i ime datoteke. Ime datoteke je definirano unutar metode pomoću koje se spremaju rezultati obrade fotografije.

```
class Plotter():
        ---odrezano---
    def plot_dark_lines(self,file):
        with open(file, 'r') as dict_points:
            points = json.load(dict_points)
            drawing_line_points = points['dark_points']
            move_to_new_line_points = points['move_to_new_line_dark']
        for l in drawing_line_points:
            self.pen_up()
            print('Going to line: ', self.count_line2 + 1, '/',
    len(drawing_line_points))
            for xy in move_to_new_line_points[self.count_line2]:
                print('X_Y: ', xy)
                self.current_x = xy[0]
                self.current_y = xy[1]
                self.get_position_xy()
            self.first2 = True
            for x_y in l:
                print('Drawing line: ', self.count_line2 + 1, '/',
    len(drawing_line_points))
                print('X_Y: ', x_y)
                if self.first2:
                    self.pen_down()
                    self.first2 = False
                self.current_x = x_y[0]
                self.current_y = x_y[1]
                self.get_position_xy()
            self.count_line2 += 1
        print('Line :', self.count_line2, '/', len(drawing_line_points), ' drawn')
```

Slika 4.42. *Python* kod metode za ispisivanje okomitih linija koje predstavljaju tamne tonove obrađene fotografije unutar razreda Plotter

Unutar metode se prije svega otvara i čita sadržaj unutar datoteke preko metode *open()*, gdje je prvi ulaz putanja do datoteke, dok je drugi ulaz string *'r'* koji označava čitanje sadržaja datoteke. Nakon toga se pomoću metode iz *json* biblioteke *json.load()* učitani

tekstualni sadržaj pretvara iz JSON formata u *Python* rječnik spremljen u varijabli *points.* Budući da ova metoda ima ulogu ispisivanja okomitih linija koje predstavljaju tamne tonove, iz rezultata se preuzimaju liste točaka po kojima se manipulator giba kada ispisuje tamne tonove te lista točaka kojima se manipulator giba kako bi došao do nove linije ispisivanja. Rezultati se spremaju u dvije liste, *drawing_line_points i move_to_new_line_points.* Nakon toga se pomoću *for* petlje prvo iterira kroz listu točaka po kojima se manipulator giba kada ispisuje linije. Unutar petlje se poziva metoda *self.pen_up()* koja podiže kemijsku olovku od podloge manipulatora zato što se kemijska olovka mora prvo pomaknuti iz početnog položaja u položaj prve točke ispisa te se pri tome ne smije događati ispisivanje. Stoga se u prvoj ugniježđenoj *for* petlja iterira kroz prvu listu iz liste točaka kada se manipulator kreće od kraja trenutne do početka sljedeće linije ispisa. Unutar *for* petlje se redom zadaju točke manipulatoru te se manipulator prema njima pokreće pomoću metode *self.get_position_xy().* Kada manipulator dođe na početak linije poziva se metoda *self.pen_down()* kako bi se kemijska olovka spustila na podlogu manipulatora. U drugoj ugniježđenoj *for* petlji se iterira po točkama kojima se manipulator treba gibati za ispis linije te se manipulator pokreće i ispisuje fotografiju preko metode *self.get_position_xy().* Analogno, za ispis okomitih linija koje predstavljaju svijetle tonove fotografije je napravljena metoda *plot_light_lines(self,file).* Jedina razlika što se koriste vrijednosti rezultata obrade fotografije za točke kojima se manipulator giba kada ispisuje okomite linije svijetlih tonova te točke kojima se manipulator giba kada prelazi u iduću liniju ispisa svijetlih tonova. Budući da su metode podjednake ovu metodu nije potrebno dodatno pojasniti te je ona u cijelosti prikazana u prilogu. Kako bi unutar glavne skripte bilo moguće pozvati metode za ispisivanje fotografije jednu za drugom, potrebno je napraviti metodu s kojom se manipulator vraća u početni položaj nakon završetka jedne od metoda za ispisivanje okomitih linija obrađene fotografije te se sve vrijednosti varijabli instance ovom metodom vraćaju u vrijednosti početnog položaja. Metoda je nazvana *default_position(self)*(slika 4.43.).

```
class Plotter():
        ---odrezano---
    def default_pisition(self):
        print('going to starting position')
        self.error1 = 0
        self.error2 = 0
        self.alfa = 90
        self.beta  = 90
        self.count_line2 = 0
        self.count_line1 = 0
        self.first1 = True
        self.first2 = True
        while self.final_steps_count1 != 0 or self.final_steps_count2 != 0:
            print(self.final_steps_count1, self.final_steps_count2)
            if self.final_steps_count1 >= 1:
                self.move_anti_clockwise_1()
                self.final_steps_count1 -= 1
            if self.final_steps_count1 <= -1:
                self.move_clockwise_1()
                self.final_steps_count1 += 1
            if self.final_steps_count2 >= 1:
                self.move_anti_clockwise_2()
                self.final_steps_count2 -= 1
            if self.final_steps_count2 <= -1:
                self.move_clockwise_2()
                self.final_steps_count2 += 1
```

Slika 4.43. *Python* kod metode za povratak manipulatora u početni položaj

Unutar ove metode se mehanizam manipulatora vraća na početni položaj preko varijabli instance *self.final_steps_count1* i *self.final_steps_count2*, definiranih unutar metode *self.get_position_xy,* koje pamte koliko je koraka svaki motor napravio tijekom ispisa linija. Prema konačnom iznosu varijabli instance *self.final_steps_count1 i self.final_steps_count2* se motori pomiču mehanizam u početni položaj. Također, sve varijable koje se odnose na matematički model manipulatora se postavljaju u vrijednosti početnog položaja.

5. EKSPERIMENTALNI REZULTATI ISPISA FOTOGRAFIJA

Kako bi se omogućilo eksperimentalno ispisivanje obrađenih fotografija potrebno je napraviti novu *Python* skriptu (slika 5.1.) u kojoj se pozivaju sve potrebne metode iz razreda za upravljanje robotskim manipulatorom. U skriptu je prije svega potrebno unijeti razred *Plotter* iz skripte u kojoj nalazi.

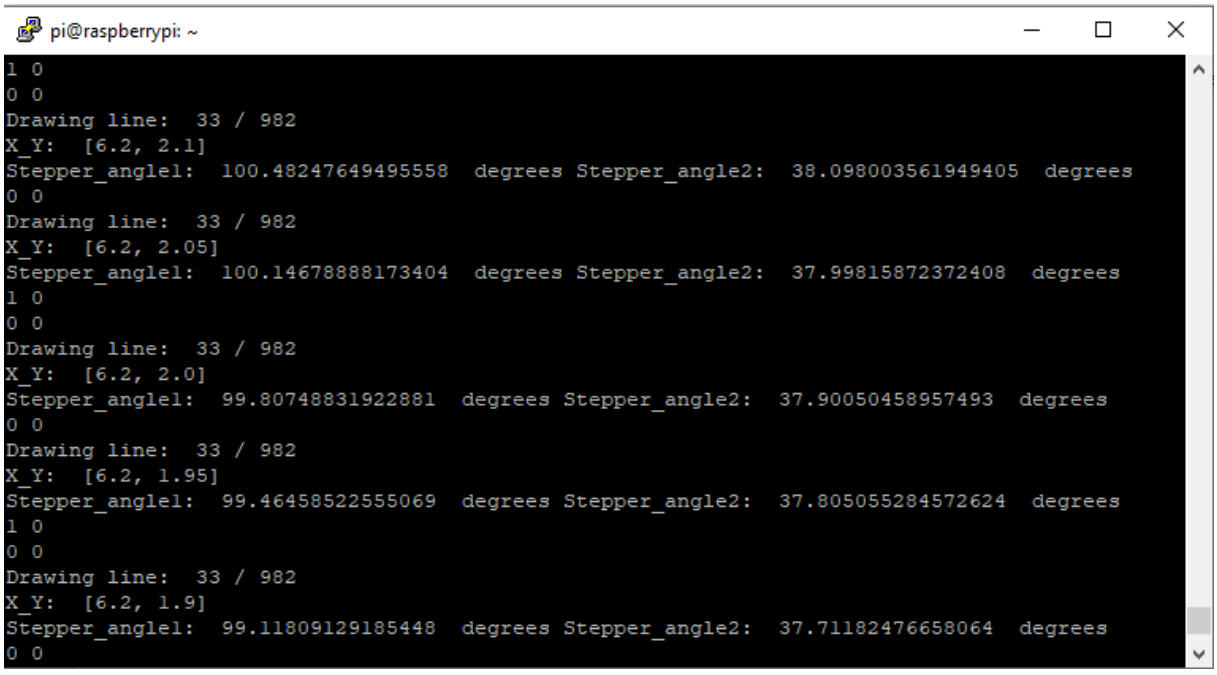
```
from Plotter import *

plotter = Plotter()
plotter.move_to_start_point()
txt_file = input('Enter_image_name:')
txt_file = str(txt_file)
plotter.plot_dark_lines(txt_file)
plotter.pen_up()
plotter.default_pisition()
plotter.plot_light_lines(txt_file)
plotter.pen_up()
plotter.default_pisition()
```

Slika 5.1. *Python* kod glavne skripte za ispisivanje obrađene fotografije

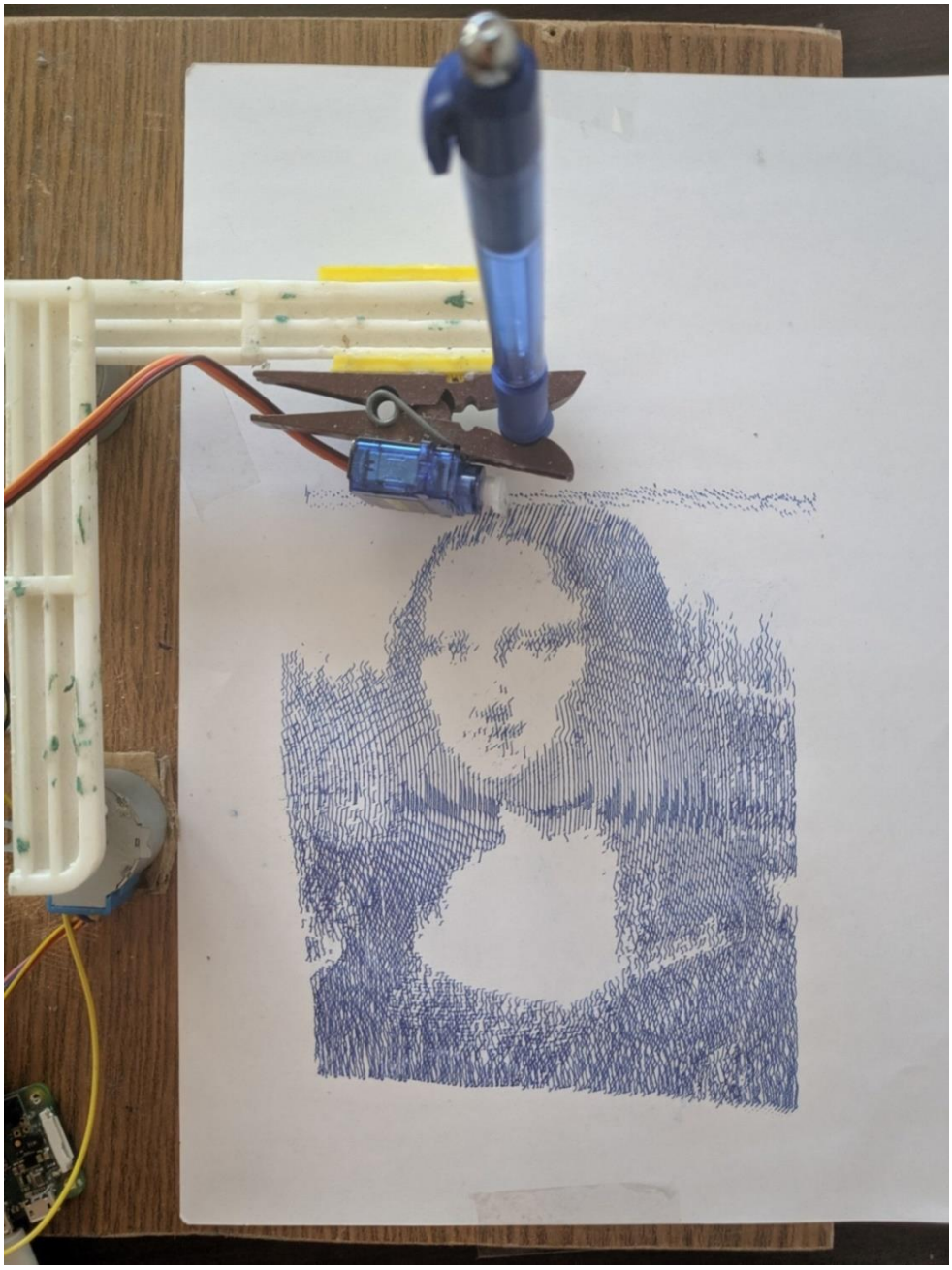
Unutar skripte se stvara novi objekt *plotter* s razredom *Plotter.* Nakon stvaranja objekta se poziva metoda *plotter.move_to_start_point()* kojom se sami manipulator postavlja u početni položaj pomoću tipkovnice računala. Kada se manipulator postavi u početni položaj, na podlogu manipulatora se postavlja papir po kojem će manipulator ispisivati fotografiju. Papir je potrebno fiksirati na podlogu kako nebi došlo do pomicanja tokom ispisa fotografije. Također je potrebno učvrstiti kemijsku olovku na odgovarajući držač za kemijsku olovku te postaviti plastični nastavak na vratilo rotora servo motoru na način da kemijska olovka ne dira površinu papira prije ispisivanja fotografije. Prije nego što se pozovu ostale metode, korisnika se traži da upiše ime datoteke s rezultatima obrade fotografije koja je prethodno poslana na *Raspberry Pi*. Naziv datoteke je definiran unutar metode za spremanje rezultata obrade fotografije. Nakon unosa ulaza pozivaju se redom metode. Prva metoda je metoda *plotter.plot_dark_lines(txt_file)* kojom manipulator ispisuje okomite linije tamnih tonova fotografije. Nakon što manipulator ispiše sve linije koje predstavljaju tamne tonove fotografije, poziva se metoda *plotter.pen_up()* kako bi podigli kemijsku olovku od papira nakon ispisivanja. Nakon što se kemijska olovka podigne od podloge, poziva se metoda *plotter.default_position()* koja

manipulator vraća u početni položaj. Kada se manipulator vrati u početni položaj poziva se metoda *plotter.plot_dark_lines(txt_file)* kojom se ispisuju svijetli tonovi fotografije. Kada se ispišu sve linije koje predstavljaju svijetle tonove ponovno se poziva metoda kojom se manipulator vraća u početni položaj. S ovom metodama završava ispis fotografije.

Stvorenu glavnu skriptu je potrebno poslati na *Raspberry Pi* preko *SCP* protokola unutar naredbenog retka računala kao u odjeljku gdje se testira matematički model manipulatora. Skripta se pokreće u naredbenom retku *Raspberry Pi-a* kojem smo pristupili preko *SSH* mrežnog protokola aplikacijom *PuTTy*. Nakon pokretanja skripte događa se ispisivanje fotografije te se u naredbenom prozoru (slika 5.2) *Raspberry Pi-a* prikazuje položaj kemijske olovke u koordinatnom sustavu manipulatora te redni broj linije koju manipulator trenutno ispisuje u odnosu na ukupan broj linija ispisivanja. Također se može vidjeti i kut koji ruke manipulatora zauzimaju. Eksperimentalno su ispisane fotografije koje su obrađene u poglavlju o programskoj podršci obrade fotografije.



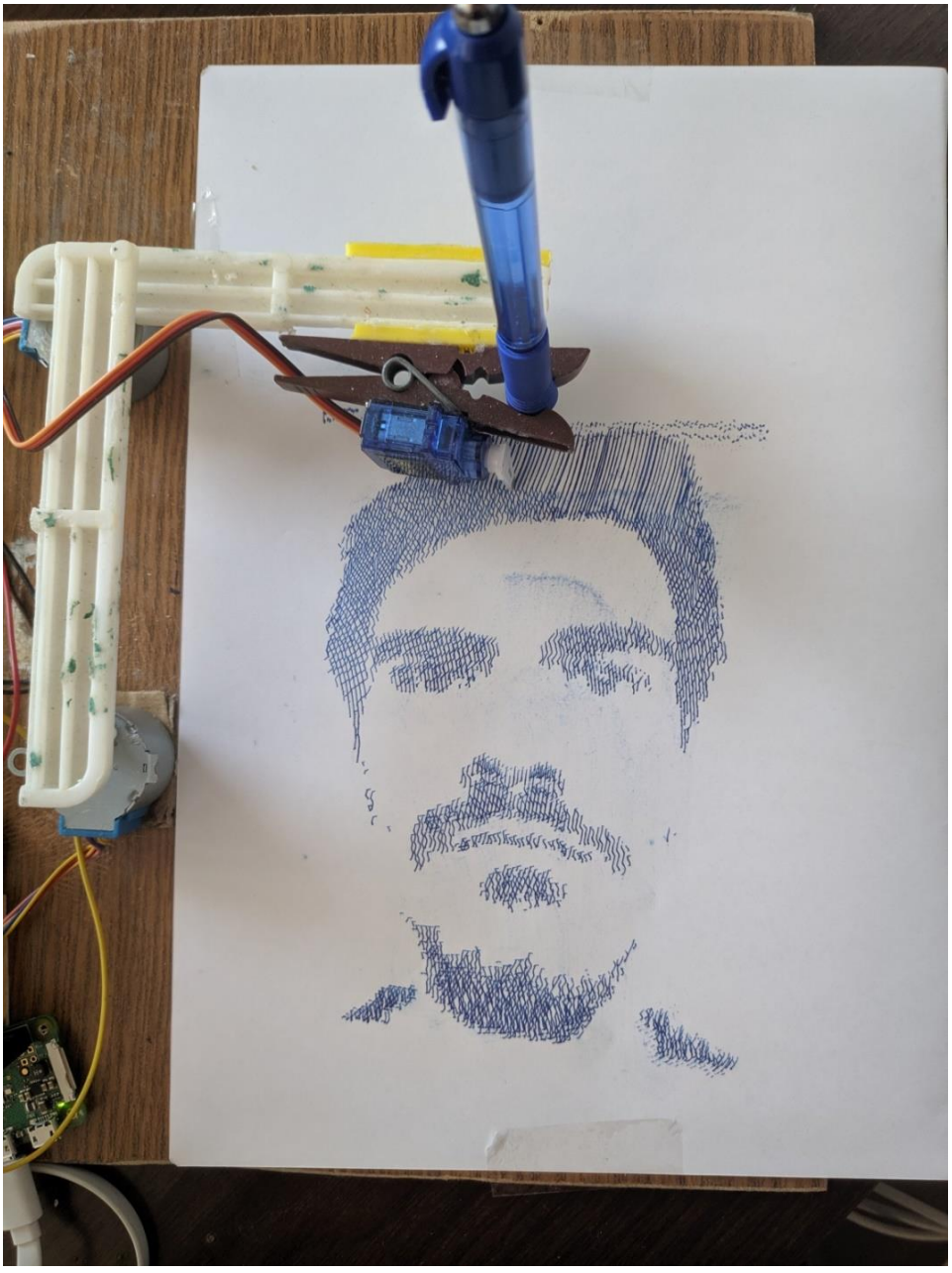
Slika 5.2. Naredbeni prozor *Raspberry Pi-a* tijekom ispisivanja fotografije



Slika 5.3. Eksperimentalni rezultat ispisa umjetničkog djela *Mona Lisa*

Na slici 5.3. je prikazan rezultat eksperimentalnog ispisa umjetničkog djela *Mona Lis,* dok je na slici 5.4. prikazan rezultat eksperimentalnog ispisa fotografije autora diplomskog rada. Samo ispisivanje fotografije može trajati između 40-60 minuta, ovisno o veličini fotografije te broju linija koje manipulator mora ispisati. Vrijeme ispisivanja se nije uspjelo ubrzati zbog nedostataka u konstrukciji manipulatora. Na slikama se može vidjeti da je sami manipulator uspješno izveden. Usporede li se rezultati obrade fotografije s rezultatima eksperimentalnog ispitivanja manipulatora može se primijetiti da manipulator ne ispisuje najtočnije linije. Linije bi trebale biti u potpunosti ravne, ali

nisu. Uzrok je taj što koračni motori rade prevelik kut prilikom pomicanja u jednom koraku. Zbog toga mehanizam ne pomiče kemijsku olovku u pravilnim koracima zbog čega se stvara greška. Dodatan razlog zašto nastaje greška je taj što koračni motori prilikom ispisivanja fotografije mogu izgubiti korak. Jedno od rješenja za uklanjanje grešaka je korištenje senzora koji bi davao povratnu informaciju o trenutnom položaju kemijske olovke te bi se prema njoj regulirao položaj koračnih motora. Budući da je cilj dobiti umjetničku sliku, moglo bi se reći da je zbog svih nedostataka rezultat još zanimljiviji jer na neki način predstavlja umjetnički izražaj.



Slika 5.4. Eksperimentalni rezultat ispisa fotografije autora diplomskog rada

6. ZAKLJUČAK

Kroz ovaj rad je izveden niskobudžetni robotski manipulator za ispisivanje fotografije. Manipulator je napravljen od jednostavne konstrukcije koja definira mehanizam manipulatora. Mehanizam manipulatora se pokreće pomoću dva koračа motora i jednog servo motora koji su spojeni na *Raspberry Pi.* Za mehanizam manipulatora je izveden matematički model kojim se dobiva položaj kemijske olovke u koordinatnom sustavu manipulatora u odnosu na kuteve koje ruke mehanizma moraju zauzeti kako bi se mehanizam postavio u željeni položaj. Uspješno je napravljena programska podrška za upravljanje manipulatorom gdje je implementiran matematički model mehanizma manipulatora te je omogućeno pokretanje manipulatora u x-y smjeru koordinatnog sustava. Programska podrška je pisana objektno orijentiranim programiranjem u programskom jeziku *Python.* Također je napravljena programska podrška obrade fotografije. Obrada fotografije se temelji na slikarskoj tehnici sjenčanja gdje se fotografija obrađuje s okomitim linijama čija udaljenost određuje označavaju ton boje fotografije. Pomoću *Python* alata za računalnu obradu fotografije dobivene su točke koje se zadaju manipulatoru kako bi se fotografija uspješno ispisala. Eksperimentalnim testiranjem, odnosno ispisom fotografije, zaključeno je da je manipulator uspješno izveden. Kada se rezultati ispisa usporede sa rezultatima obrade fotografije, vidi se da manipulator ima dosta nedostataka koji se mogu poboljšati. Linije koje manipulator ispisuje, prema obradi fotografije, bi trebale biti ravne, ali na eksperimentalnim rezultatima se vidi da linije nisu u potpunosti ravne. Ovaj nedostatak je zapravo pozitivan, budući da je cilj bio osmisliti manipulator koji ispisuje fotografije u umjetničkom stilu. Manipulator bi prije svega davao bolje rezultate kada bi se koristili bolji koračni motori, odnosno motori koji se mogu upravljati s manjim koracima te bi se na taj način stvarala manja greška ispisivanja. Također, poboljšanjem same konstrukcije manipulatora postiglo bi se bolji rezultati te bi se omogućilo brže ispisivanje fotografije. Unatoč nedostacima, manipulator je uspješno izveden ako uzmemo u obzir cilj rada.

7. LITERATURA

[1] Raspberry Pi Foundation, „Raspberry Pi“ Raspberry Pi Foundation-About us, 5.4.2020. , https://www.raspberrypi.org/about/

[2] Raspberry Pi Foundation, „Raspberry Pi“ Buy a Raspberry Pi Zero W- Raspberry Pi, 5.4.2020, *https://www.raspberrypi.org/products/raspberry-pi-zero-w/*

[3] Raspberry Pi Foundation, „Raspberry Pi“ FAQs - Raspberry Pi Documentation, 5.4.2020., https://www.raspberrypi.org/documentation/faqs/

[4] Raspberry Pi Foundation, „ Raspberry Pi“ GPIO - Raspberry Pi Documentation, 7.4.2020., https://www.raspberrypi.org/documentation/usage/gpio/

[5] Raspberry Pi Foundation, „Raspberry Pi“ SSH using Windows - Raspberry Pi Documentation,7.4.2020.,https://www.raspberrypi.org/documentation/remote-access/ssh/windows.md

[6] Tarun, Agarwal, „Elprocus“ Stepper Motors Types, Advatages and Aplications, 2013, https://www.elprocus.com/stepper-motor-types-advantages-applications/

[7] Components 101, „Components101“ ,28BYJ-48 Stepper Motor Pinout Wiring, Specifications, Uses Guide & Datasheet,2018 .,https://components101.com/motors/28byj-48-stepper-motor

[8] Texas, Instruments. *"ULN2003A Datasheet"*., 2019

[9] Daren, Sawicz, „Hobby Servo Funamentals“ ., 2008

[10] Components 101, „Components101“ , Servo Motor SG-90 Basics, Pinout, Wire Description, Datasheet, 2017., https://components101.com/servo-motor-basics-pinout-datasheet

[10] Real Python , „Real Python“ , Object-Oriented Programming (OOP) in Python 3 – Real Python, 2019., https://realpython.com/python3-object-oriented-programming/

[11] Python Software Foundation, “Python Packaging Authority” , Python Packaging Authority — PyPA documentation, 13.4.2020., https://www.pypa.io/en/latest/

[12] Chris Hager, „RPIO 0.10.0 documentation“, Welcome to RPIO’s documentation! — RPIO 0.10.0 documentation, 2013. , https://pythonhosted.org/RPIO/#

[13] pigpio @ *abyz.me*.uk, „PiGPIO“ , pigpio library , 14.4.2020. , http://abyz.me.uk/rpi/pigpio/

[14] Python Software Foundation , "3.8.2. Python Documentation“ , time-Time access and conversion , 15.4.2020. , https://docs.python.org/3/library/time.html

[15] Miguel Angel Garcia, „GitHub“ , magmax/Python-readchar, 2015. , https://github.com/magmax/python-readchar

[16] Python Software Foundation , "3.8.2. Python Documentation“ , math-Mathematical functions, 15.4.2020. , https://docs.python.org/3/library/math.html

[17] Python Software Foundation , "3.8.2. Python Documentation“ , json –JSON encoder and decoder , 15.4.2020. , https://docs.python.org/3/library/json.html

[18] PAVKOVIĆ I D.VELJAN, Elementarna matematika 2, Školska knjiga, Zagreb, 1995.

[19] Anthony James Founder , „Linux Academy“ , SSH and SCP: Howto, tips & tricks – Linux Academy, 2012. , https://linuxacademy.com/blog/linux/ssh-and-scp-howto-tips-tricks/

[20] South, Helen , The Everything Drawing Book, Everything 2009.

[21] Christian Calviño, https://www.pinterest.com/pin/108649409734676848/

[22] John Hunter, Darren Dale, Eric Firing, Michael Droettboom, „Matplotlib“ , Matplotlib: Python plotting — Matplotlib 3.2.1 documentation, 17.4.2020. , https://matplotlib.org/

[23] Python Software Foundation , "3.8.2. Python Documentation“ , subprocess – Subprocess management, 17.4.2020. , https://docs.python.org/3/library/subprocess.html

[24]NumPy developers, „NumPy“, NumPy — NumPy, 17.4.2020. , https://numpy.org/

[25] Open CV, „ Open CV“ , Open CV Computer Vision Library, 17.4.2020. https://opencv.org/

[26] Alex Clark and Contributors Revision, „Pillow (PIL Fork)“ , Pillow — Pillow (PIL Fork) 7.1.1 documentation, 17.4.2020. , https://pillow.readthedocs.io/en/stable/#

[27] Danny Pascale, A Review of RGB Color Spaces, BabelColor, 2003.

PRILOG

1. *Python* kod programske podrške upravljanja manipulatora
2. *Python* kod programske podrške obrade fotografije
3. *Python* kod vizulizacije područja rada manipulatora bez matematičkog modela

1. Python kod programske podrške upravljanja manipulatora

Python skripta s razredom *Plotter*

```
import RPi.GPIO as GPIO
from time import sleep
import readchar
import math
import numpy
import json
import pigpio


class Plotter():
    def __init__(self):
        ### STEPPER MOTORS
        GPIO.setwarnings(False)
        GPIO.setmode(GPIO.BCM)
        self.motor_channel_2 = (6, 13, 19, 26)
        self.motor_channel_1 = (12, 16, 20, 21)
        GPIO.setup(self.motor_channel_1, GPIO.OUT)
        GPIO.setup(self.motor_channel_2, GPIO.OUT)
        self.delay = 0.01
        ### SERVO MOTORS
        self.servo_pin = 3
        self.rpi = pigpio.pi()
        self.rpi.set_PWM_frequency(self.servo_pin, 50)
        self.rpi.set_servo_pulsewidth(self.servo_pin, 1500)
        ### Početni položaj manipulatora
        self.arm_1 = 9.2   # Stvaran dužina unutarnje ruke izmjerena na
konstrukciji[cm]
        self.arm_2 = 10.5  # Stvarna dužina vanjske ruke izmjerena na
konstrukciji[cm]
        self.current_x = 10.5   # Trenutni x prema matematičkom modelu[cm]
        self.current_y = 9.2   # Trenutni y prema matematičkom modelu[cm]
        self.alfa = 90  # Trenutni/početni kut prvog koračnog motroa
        self.beta = 90  # Trenutni/početni kut drugog koračnog motroa

        #Stupanj za koji se koračni motor rotira za jedan korak
        self.step_angle = 0.703125
        # Greška koju manipulator stvara zbog prevelikog stupnja pomaka koračnog
motra
        self.error1 = 0  # Greška za prvi motor
        self.error2 = 0  # Greška za drugi motor
        # Brojač broja koraka koje manipulatori naprave
        self.final_steps_count1 = 0  # Ukupan broj koraka prvog motora
        self.final_steps_count2 = 0  # Ukupan broj koraka drugog motora
        # Broj koraka koje manipulator napravi da dođe iz jednog položaja u drugi
        self.round_steps_count1 = 0  # Broj koraka prvog koračnog motora
        self.round_steps_count2 = 0 # Broj koraka drugog koračnog motora
        # Granice rada manipulatora
        self.bounds = (6, -8, 17, 8)
        # Brojač linija
```

```
        self.count_line2 = 0
        self.count_line1 = 0
        # Okidač
        self.first1  = True
        self.first2 = True

    ### METODA ZA PODIZANJE KEMIJSKE OLOVKE OD PODLOGE MANIPULATORA
    def pen_up(self):
        self.rpi.set_servo_pulsewidth(self.servo_pin, 1500)
        sleep(0.1)
    ### METODA ZA SPUŠTANJE KEMIJSKE OLOVKE NA PODLOGU MANIPULATORA
    def pen_down(self):
        self.rpi.set_servo_pulsewidth(self.servo_pin, 1000)
        sleep(0.1)

    ### mETODA ZA POKRETANJE PRVOG KORAČNOG MOTORA U OBRNUTOM SMJERU KAZALJKE NA
SATU
    def move_anti_clockwise_1(self):
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        sleep(self.delay)

    ### METODA ZA POKRETANJE DRUGOG KORAČNOG MOTORA U SMJERU KAZALJKE NA SATU
    def move_clockwise_2(self):
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        sleep(self.delay)

    ### METODA ZA POKRETANJE PRVOG KORAČNOG MOTORA U SMJERU KAZALJKE NA SATU
    def move_clockwise_1(self):
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
```

```
GPIO.LOW))
        sleep(self.delay)

    ### METODA ZA POKRETANJE DRUGOG KORAČNOG MOTORA U OBRNUTOM SMJERU KAZALJKE NA
SATU
    def move_anti_clockwise_2(self):
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        sleep(self.delay)

    ### METODA ZA POKRETANJE PRVOG KORAČNOG MOTORA U OBRNUTOM SMJERU KAZALJE NA
SATU I DRUGOG MOTORA U SMJERU KAZALJKE NA SATU
    def move_anti1_clock2(self):
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        sleep(self.delay)

    ### METODA ZA POKRETANJE PRVOG KORAČNOG MOTORA U SMJERU KAZALJE NA SATU I
DRUGOG MOTORA U SMJERU OBRNUTOM OD KAZALJKE NA SATU
    def move_clock1_anti2(self):
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
```

```
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        sleep(self.delay)

    ### METODA ZA POKRETANJE KORAČNIH MOTORA U OBRNUTOM SMJERU KAZALJKE NA SATU
    def move_anti_clockwise_12(self):
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        sleep(self.delay)

    ### METODA ZA POKRETANJE OBA KORAČNA MOTORA U SMJERU KAZALJKE NA SATU
    def move_clocwise_12(self):
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.LOW, GPIO.LOW,
GPIO.HIGH))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        GPIO.output(self.motor_channel_2, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH,
GPIO.LOW))
        sleep(self.delay)
        GPIO.output(self.motor_channel_1, (GPIO.HIGH, GPIO.HIGH, GPIO.LOW,
GPIO.LOW))
        GPIO.output(self.motor_channel_2, (GPIO.LOW, GPIO.LOW, GPIO.HIGH,
GPIO.HIGH))
        sleep(self.delay)

    ### METODA ZA POKRETANJE KORAČNIH MOTORA U ODNOSU NA IZRAČUNATI BROJ KORAKA
    def move_motors_by_step_count(self):
        while self.round_steps_count1 != 0 or self.round_steps_count2 != 0:
```

```
            if self.round_steps_count1 >= 1 and self.round_steps_count2 >= 1:
                self.move_clocwise_12()
                self.round_steps_count1 -= 1
                self.round_steps_count2 -= 1
                print(self.round_steps_count1,self.round_steps_count2)
            elif self.round_steps_count1 <= -1 and self.round_steps_count2 >= 1:
                self.move_anti1_clock2()
                self.round_steps_count1 += 1
                self.round_steps_count2 -= 1
                print(self.round_steps_count1, self.round_steps_count2)
            elif self.round_steps_count1 <= -1 and self.round_steps_count2 <= -1:
                self.move_anti_clockwise_12()
                self.round_steps_count1 += 1
                self.round_steps_count2 += 1
                print(self.round_steps_count1, self.round_steps_count2)
            elif self.round_steps_count1 >= 1 and self.round_steps_count2 <= -1:
                self.move_clock1_anti2()
                self.round_steps_count1 -= 1
                self.round_steps_count2 += 1
                print(self.round_steps_count1, self.round_steps_count2)
            elif self.round_steps_count1 >= 1:
                self.move_clockwise_1()
                self.round_steps_count1 -= 1
                print(self.round_steps_count1, self.round_steps_count2)
            elif self.round_steps_count1 <= -1:
                self.move_anti_clockwise_1()
                self.round_steps_count1 += 1
                print(self.round_steps_count1, self.round_steps_count2)
            elif self.round_steps_count2 >= 1:
                self.move_clockwise_2()
                self.round_steps_count2 -= 1
                print(self.round_steps_count1, self.round_steps_count2)
            elif self.round_steps_count2 <= -1:
                self.move_anti_clockwise_2()
                self.round_steps_count2 += 1
                print(self.round_steps_count1, self.round_steps_count2)

    def get_position_xy(self):


        ### IMPLEMENTACIJA MATEMATIČKOG MODELA

        omega_x = math.atan(self.current_y / self.current_x)
        AC = self.current_x / math.cos(omega_x)
        gama = math.acos((AC ** 2 + self.arm_1 ** 2 - self.arm_2 ** 2) / (2 *
self.arm_1 * AC))
        new_alfa = omega_x +  gama
        new_alfa_degrees = math.degrees(new_alfa)
        epsilon = math.acos((self.arm_1 ** 2 + self.arm_2 ** 2 - AC ** 2) / (2 *
self.arm_1 * self.arm_2))
        new_beta = epsilon
        new_beta_degrees = math.degrees(new_beta)
        ### RAČUNAJE BROJA KORAKA KOJE MOTOR TREBA NAPRAVIT DA ZAUZME NOVI KUT
        delta_1 = self.alfa - new_alfa_degrees
        self.alfa = new_alfa_degrees
        delta_2 = self.beta - new_beta_degrees
        self.beta = new_beta_degrees
        print('Stepper_angle1: ', self.alfa, ' degrees', 'Stepper_angle2: ',
```

```
                self.beta, ' degrees')
        count_steps1 = delta_1 / self.step_angle
        count_steps2 = delta_2 / self.step_angle
        count_steps1 += self.error1
        count_steps2 += self.error2
        self.round_steps_count1 = round(count_steps1)
        self.round_steps_count2= round(count_steps2)
        self.final_steps_count1 += self.round_steps_count1
        self.final_steps_count2 += self.round_steps_count2
        self.error1 = count_steps1 - self.round_steps_count1
        self.error2 = count_steps2 - self.round_steps_count2
        print(self.round_steps_count1,self.round_steps_count2)
        #POKRETANJE MOTORA
        self.move_motors_by_step_count()


    ### METODA ZA UPRAVLJANJE MANIPULATOROM U X-Y KOORDINATNOM SUSTAVU
MANIPULATORA
    def drive_xy(self):
        while True:
            key = readchar.readchar()
            if key == "q":
                return
            elif key == "w":
                self.current_y = self.current_y + 0.1
            elif key == "s":
                self.current_y = self.current_y - 0.1
            elif key == "d":
                self.current_x = self.current_x + 0.1
            elif key == "a":
                self.current_x = self.current_x - 0.1
            print('X_Y: ', round(self.current_x), round(self.current_y))
            self.get_position_xy()



    ### METODA ZA DOBIVANJE LINIJA GRANICA MANIPULATORA
    def get_box(self):
        start_point = (self.bounds[0], self.bounds[1])
        line = list(zip(numpy.linspace(self.current_x, start_point[0], 50),
                        numpy.linspace(self.current_y, start_point[1], 50)))
        print(line)
        for x_y in line:
            self.current_x = x_y[0]
            self.current_y = x_y[1]
            self.get_position_xy()

    ### METODA ZA ISPISIVANJE GRANICA MANIPULATORA ODREĐENOG PRAVOKUTNIKOM U KOJEM
JE OMOGUĆEN ISPIS
    def draw_box(self):
        self.pen_up()
        self.get_box()
        self.pen_down()
        for y in numpy.arange(self.bounds[1], self.bounds[3], 0.05):
            self.current_y = y
            self.get_position_xy()
        for x in numpy.arange(self.bounds[0], self.bounds[2], 0.05):
            self.current_x = x
```

```
            self.get_position_xy()
        for y in numpy.arange(self.bounds[3], self.bounds[1], -0.05):
            self.current_y = y
            self.get_position_xy()
        for x in numpy.arange(self.bounds[2], self.bounds[0], -0.05):
            self.current_x = x
            self.get_position_xy()

    ###METODA ZA  ISPISIVANJE TAMNIH TONOVA FOTOGRAFIJE
    def plot_dark_lines(self,file):

        with open(file, 'r') as dict_points:
            points = json.load(dict_points)
            drawing_line_points = points['dark_points']
            move_to_new_line_points = points['move_to_new_line_dark']

        for l in drawing_line_points:
            self.pen_up()
            print('Going to line: ', self.count_line2 + 1, '/',
len(drawing_line_points))
            for xy in move_to_new_line_points[self.count_line2]:
                print('X_Y: ', xy)
                self.current_x = xy[0]
                self.current_y = xy[1]
                self.get_position_xy()
            self.first2 = True
            for x_y in l:
                print('Drawing line: ', self.count_line2 + 1, '/',
len(drawing_line_points))
                print('X_Y: ', x_y)
                if self.first2:
                    self.pen_down()
                    self.first2 = False
                self.current_x = x_y[0]
                self.current_y = x_y[1]
                self.get_position_xy()

            self.count_line2 += 1
        print('Line :', self.count_line2, '/', len(drawing_line_points), ' drawn')

    ### METODA ZA CRTANJE SVIJETLIH TONOVA FOTOGRAFIJE
    def plot_light_lines(self,file):

        with open(file, 'r') as dict_points:
            points = json.load(dict_points)
            drawing_line_points = points['light_points']
            move_to_new_line_points = points['move_to_new_line_light']

        for l in drawing_line_points:
            self.pen_up()
            print('Going to line: ', self.count_line2 + 1,'/',
len(drawing_line_points))
            for xy in move_to_new_line_points[self.count_line2]:
                print('X_Y: ', xy)
                self.current_x = xy[0]
                self.current_y = xy[1]
                self.get_position_xy()
            self.first2 = True
```

```
            for x_y in l:
                print('Drawing line: ', self.count_line2 + 1, '/',
len(drawing_line_points))
                print('X_Y: ', x_y)
                if self.first2:
                    self.pen_down()
                    self.first2 = False
                self.current_x = x_y[0]
                self.current_y = x_y[1]
                self.get_position_xy()

            self.count_line2 += 1
        print('Line :', self.count_line2,'/', len(drawing_line_points) ,' drawn')

    ### METODA ZA POSTAVLJANJE MANIPULATORA U POČETNI POLOŽAJ PRIJE POKRETANJA
ISPISIVANJA
    def move_to_start_point(self):

        while True:
            key = readchar.readchar()

            if key == "e":
                self.move_clockwise_1()
                print('Moving stepper_1 anticlockwise')
            elif key == "w":
                self.move_anti_clockwise_1()
                print('Moving stepper_1 clockwise')
            elif key == "s":
                self.move_anti_clockwise_2()
                print('Moving stepper_2 anticlockwise')
            elif key == "d":
                self.move_clockwise_2()
                print('Moving stepper_2 clockwise')
            elif key == 't':
                self.pen_up()
                print('Pen up!')
            elif key =='g':
                self.pen_down()
                print('Pen down!')
            elif key == "q":
                break

    ### METODA ZA VRAČANJE MANIPULATORA U POČETNI POLOŽAJ
    def default_pisition(self):
        print('going to starting position')
        self.pen_up()
        self.error1 = 0
        self.error2 = 0
        self.alfa = 90
        self.beta  = 90
        self.count_line2 = 0
        self.count_line1 = 0
        self.first1 = True
        self.first2 = True
        while self.final_steps_count1 != 0 or self.final_steps_count2 != 0:
            print(self.final_steps_count1, self.final_steps_count2)
            if self.final_steps_count1 >= 1:
                self.move_anti_clockwise_1()
```

```
            self.final_steps_count1 -= 1
        if self.final_steps_count1 <= -1:
            self.move_clockwise_1()
            self.final_steps_count1 += 1
        if self.final_steps_count2 >= 1:
            self.move_anti_clockwise_2()
            self.final_steps_count2 -= 1
        if self.final_steps_count2 <= -1:
            self.move_clockwise_2()
            self.final_steps_count2 += 1
```

Glavna Python skripta za testiranje matematičkog modela.

```
from Plotter import *

plotter = Plotter()
plotter.move_to_start_point()
plotter.drive_xy()
```

Glavna *Python* skripta za ispisivanje fotografije.

```
from Plotter import *

plotter = Plotter()
plotter.move_to_start_point()
txt_file = input('Enter_image_name:')
txt_file = str(txt_file)
plotter.plot_dark_lines(txt_file)
plotter.pen_up()
plotter.default_pisition()
plotter.plot_light_lines(txt_file)
plotter.pen_up()
plotter.default_pisition()
```

2. *Python* kod programske podrške obrade fotografije

Skripta s razredom *Image_processing i* dodatnim statičkim funkcijama.

```
from numpy import interp
import json
import numpy as np
import math
from PIL import Image
import cv2
import matplotlib.pyplot as plt
import subprocess

###### STATIČKE FUNKCIJE ###############

# FUNKCIJA ZA ZAOKRUŽIVANJE BROJEVA NA ODREĐENI DECIMALNI BROJ
def round_nearest(x, a):
    return round(round(x / a) * a, -int(math.floor(math.log10(a))))

# FUNKCIJA ZA ROTIRANJE FOTOGRAFIJE
preuzeta sa interneta
https://www.pyimagesearch.com/2017/01/02/rotate-images-correctly-with-opencv-and-
python/
def rotate_bound(image, angle):
    # grab the dimensions of the image and then determine the
    # center
    (h, w) = image.shape[:2]
    (cX, cY) = (w / 2, h / 2)

    # grab the rotation matrix (applying the negative of the
    # angle to rotate clockwise), then grab the sine and cosine
    # (i.e., the rotation components of the matrix)
    M = cv2.getRotationMatrix2D((cX, cY), -angle, 1.0)
    cos = np.abs(M[0, 0])
    sin = np.abs(M[0, 1])

    # compute the new bounding dimensions of the image
    nW = int((h * sin) + (w * cos))
    nH = int((h * cos) + (w * sin))

    # adjust the rotation matrix to take into account translation
    M[0, 2] += (nW / 2) - cX
    M[1, 2] += (nH / 2) - cY

    # perform the actual rotation and return the image
    return cv2.warpAffine(image, M, (nW, nH))

# FUNKCIJA ZA BRISANJE MALIH LINIJA
def remove_small_lines(list):
    new_round_lines= []
    for i in list:
        if len(i) > 1:
            new_round_lines.append(i)

    return new_round_lines

# FUNKCIJA ZA ZAOKRUŽIVANJE TOČAKA
```

```
def round_points(list, number_of_decimal_x , number_of_decimal_y):
    round_points = []
    for line in list:
        round_line = []
        for x_y in line:
            xy = []
            x = round_nearest(x_y[0], number_of_decimal_x)
            y = round_nearest(x_y[1], number_of_decimal_y)
            xy.append(x)
            xy.append(y)
            round_line.append(xy)
        round_points.append(round_line)
    return round_points

# FUNCKIJA ZA BRISANJE ISTIH LINIJA KOJE POSJEDUJU JEDAKU PRVU TOČKU
def remove_same_lines_x(list, get_round_number_of_x):
    main_lines = []
    check_point_x = 0
    first_line = False
    for line in list:
        new_point_x = int(line[0][0] * get_round_number_of_x)
        if first_line:
            if new_point_x != check_point_x:
                main_lines.append(line)
        first_line = True
        check_point_x= int(line[0][0]* get_round_number_of_x)

    return main_lines

# FUNKCIJA ZA DOBIVANJE VIŠE LINIJA AKO JE KOJIM SLUČAJEM RAZLIKA VRIJEDNOSTI Y
VEĆA OD 0.05
def split_lines(list):
    new_lines= []
    for line in list:
        check_y = 0
        first = False
        lines = []
        for points in line:
            x = points[0]
            y =int(points[1]*100)
            if first:
                if (check_y - y)**2 > 50:
                    new_lines.append(lines)
                    lines = []
            check_y = y
            new_point = [x, round_nearest(y/100,0.05)]
            lines.append(new_point)
            first = True
        new_lines.append(lines)

            first = True
        new_lines.append(lines)

    return new_line

# FUNCKIJA ZA BRISANJE ISTIH TOČAKA UNUTAR LINIJE
def remove_same_points_in_line(lines, get_round_number_of_x ,
get_round_number_of_y,number_of_decimal_x , number_of_decimal_y ):
```

```
    smooth_lines= []
    for line in lines:
        check_hor_point_x = 0
        check_hor_point_y = 0
        better_lines = []
        first_point = []
        x_first = line[0][0]
        y_first = line[0][1]
        first_point.append(x_first)
        first_point.append(y_first)
        better_lines.append(first_point)
        first = True
        new = True
        for points in line:
            if new:
                x_first = int(x_first * get_round_number_of_x)
                y_first = int(y_first * get_round_number_of_y)
            new_point = [0, 0]
            x = points[0]
            y = points[1]
            x = int(x * get_round_number_of_x)
            y = int(y * get_round_number_of_y)
            if first :
                if x != x_first or y != y_first:
                    if (x - first_point[0]) != 0  or  (y- first_point[1]) != 0 :
                        new_point[0] = round_nearest(x / get_round_number_of_x,
number_of_decimal_x)
                        new_point[1] = round_nearest(y / get_round_number_of_y,
number_of_decimal_y)
                        check_hor_point_x = x
                        check_hor_point_y = y
                        better_lines.append(new_point)
                        first = False
                        new = False
                    else:
                        check_hor_point_x = x
                        check_hor_point_y = y
                        new = False
                        first = False

                else:
                    new = False
            else:
                if (x - check_hor_point_x) != 0 or (y- check_hor_point_y) != 0 :
                    new_point[0] = round_nearest(x / get_round_number_of_x,
number_of_decimal_x)
                    new_point[1] = round_nearest(y / get_round_number_of_y,
number_of_decimal_y)
                    check_hor_point_x = x
                    check_hor_point_y = y
                    better_lines.append(new_point)
                else:
                    check_hor_point_x = x
                    check_hor_point_y = y

        smooth_lines.append(better_lines)

    return smooth_lines
```

```
class Image_processing():
    def __init__(self,photo):
        self.photo = photo
        # Vrijednosti na koje ukazuju na kojoj su udaljenosti paralelene linije
ispisivanja
        self.width_between_line_light = 0.1   # za svijetle tonove 0.1 cm
        self.width_between_line_dark = 0.05  # za tamne tonove 0.05 cm
        self.width_y = 0.05
        # Brojka s kojom dobivamo cijeli broj zaokruženih točaka gibanja
manipulatora
        self.get_round_01 = 10
        self.get_round_005 = 100
        # Pragovi tonovoa za koje manipulator ispisuje fotografiju
        self.threshold_dark = 75
        self.threshold_light = 110
        #Granice pravokutnika unutar kojih manipulator ispisjue fotografije
        self.bounds = [5, -8, 18, 11]
        # Liste u koje spremamo vrijedosti piksela
        self.Vertical_light_lines = [] # Liste točaka koje predstavljaju tamne
tonove u koordinatno sustavu fotografije
        self.Vertical_dark_lines = []
        # Liste u koje se spremaju transformirane točke
        self.Vertical_light_lines_transformed = []
        self.Vertical_dark_lines_transformed = []
        # Liste u koje se spremaju zaokružene točke u koordinatnom sustavu
manipulatora
        self.Vertical_dark_lines_round = []
        self.Vertical_light_lines_round = []
        # Starting pen position
        self.default_position = [10.5, 9.2]
        # Defining list for point which defines movement of manipulator when not
drawing
        self.Move_to_line_dark = []
        self.Move_to_line_light = []

    def import_image(self):
        self.image = cv2.imread(self.photo,cv2.IMREAD_GRAYSCALE )
        (self.height, self.width) = self.image.shape[:2]
        # Rotiranje ako je širina veća od visine fotografije
        if self.width > self.height:
            self.image= rotate_bound(self.image, angle=-90)
            (self.height, self.width) = self.image.shape[:2]
            cv2.imshow('Imported Image',self.image)
            cv2.waitKey(0)
            cv2.destroyAllWindows()
        else:
            cv2.imshow('Imported Image', self.image)
            cv2.waitKey(0)
            cv2.destroyAllWindows()
        cv2.imwrite('gray.jpg', self.image)


    def upper_frame(self):
        threshold_line_dark = 0
        threshold_line_light = self.threshold_light - 10
        thickness_line_dark = int(self.height/65)
        thickness_line_light = thickness_line_dark*2
```

```
        start_point1 = (0, 0)
        end_point1 = (self.width, 0)
        color1 = (threshold_line_dark,threshold_line_dark , threshold_line_dark)
        color2 = (threshold_line_light, threshold_line_light,
threshold_line_light)
        start_point2 = (0, int((thickness_line_dark + thickness_line_light) / 2))
        end_point2 = (self.width, int((thickness_line_dark + thickness_line_light)
/ 2))
        self.line_image = cv2.line(self.image, start_point1, end_point1, color1,
thickness_line_light)
        self.line_image = cv2.line(self.line_image, start_point2, end_point2,
color2, thickness_line_dark)
        self.framed_photo = self.photo[:-4] + '_line.jpg'
        cv2.imwrite(self.framed_photo, self.line_image)
        cv2.imshow('line_image', self.line_image)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    def define_bounds(self):
        # getting new bound (because of scaling)
        self.bounds[3] = round((self.bounds[1] + (self.bounds[2] - self.bounds[0])
* (self.height / self.width)), 1)
        if self.bounds[3] > 8:
            self.bounds[3] = 8
            self.bounds[2] = round(self.bounds[0] + (self.bounds[3] -
self.bounds[1]) * self.width / self.height)
        self.bounds[3] += 1.3
        if (self.height / self.width) * 10 > 14:
            print('Please import photo where ratio of height and width of image is
not bigger then 1.4')


        print('Bounds for this image is: ', self.bounds)

    def getting_points_pixels(self):
        self.pixel_image = Image.open(self.framed_photo)
        self.pixel_image = self.pixel_image.convert("L")
        self.pixels = self.pixel_image.load()
        for x0 in range(self.width):
            line = []
            line_light = []
            for y0 in range(self.height):
                if self.pixels[x0, y0] <= self.threshold_dark:
                    line.append([x0, y0])
                if self.pixels[x0, y0] > self.threshold_dark and self.pixels[x0,
y0] <= self.threshold_light:
                    line_light.append([x0, y0])
            self.Vertical_dark_lines.append(line)
            self.Vertical_light_lines.append(line_light)

    def image_transformation(self):
        for line in self.Vertical_dark_lines:
            xy = []
            for points in line:
                x = interp(points[0], [0, self.width], [self.bounds[0],
self.bounds[2]])
                y = interp(points[1], [0, self.height], [self.bounds[3],
self.bounds[1]])
```

```
                x_y = [x, y]
                xy.append(x_y)
            self.Vertical_dark_lines_transformed.append(xy)
        for line in self.Vertical_light_lines:
            xy = []
            for points in line:
                x = interp(points[0], [0, self.width], [self.bounds[0],
self.bounds[2]])
                y = interp(points[1], [0, self.height], [self.bounds[3],
self.bounds[1]])
                x_y = [x, y]
                xy.append(x_y)
            self.Vertical_light_lines_transformed.append(xy)

    def round_drawing_points(self):
        for line in self.Vertical_dark_lines_transformed:
            round_line = []
            for x_y in line:
                xy = []
                x = round_nearest(x_y[0], self.width_between_line_dark)
                y = round_nearest(x_y[1], self.width_y)
                xy.append(x)
                xy.append(y)
                round_line.append(xy)
            self.Vertical_dark_lines_round.append(round_line)
        for line in self.Vertical_light_lines_transformed:
            round_line = []
            for x_y in line:
                xy = []
                x = round_nearest(x_y[0], self.width_between_line_light)
                y = round_nearest(x_y[1], self.width_y)
                xy.append(x)
                xy.append(y)
                round_line.append(xy)
            self.Vertical_light_lines_round.append(round_line)

    def drawing_points_processing(self):

        Removed_same_lines_in_column_dark =
remove_same_lines_x(self.Vertical_dark_lines_round, self.get_round_005)
        Removed_same_lines_in_column_light =
remove_same_lines_x(self.Vertical_light_lines_round,self.get_round_01 )

        Splitting_lines_from_one_row_dark =
split_lines(Removed_same_lines_in_column_dark)
        Splitting_lines_from_one_row_light =
split_lines(Removed_same_lines_in_column_light)


        Removed_small_lines_dark =
remove_small_lines(Splitting_lines_from_one_row_dark)
        Removed_small_lines_light=
remove_small_lines(Splitting_lines_from_one_row_light)

        Removed_same_points_in_one_line_dark =
remove_same_points_in_line(Removed_small_lines_dark,
self.get_round_005,
self.get_round_005,
```

```
self.width_between_line_dark,
self.width_y)


        Removed_same_points_in_one_line_light =
remove_same_points_in_line(Removed_small_lines_light,
self.get_round_01,
self.get_round_005,
self.width_between_line_light,
self.width_y)

        self.Final_lines_dark =
remove_small_lines(Removed_same_points_in_one_line_dark)
        self.Final_lines_light
=remove_small_lines(Removed_same_points_in_one_line_light)

    def get_move_to_new_drawing_line(self):

        x = self.default_position[0]
        y = self.default_position[1]
        for line in self.Final_lines_dark:
            first_point = line[0]
            to_new_line = list(zip(np.linspace(x, first_point[0], 100),
                                   np.linspace(y, first_point[1], 100)))
            self.Move_to_line_dark.append(to_new_line)
            for points in line:
                x= points[0]
                y= points[1]

        x = self.default_position[0]
        y = self.default_position[1]
        for line in self.Final_lines_light:
            first_point = line[0]
            to_new_line = list(zip(np.linspace(x, first_point[0], 100),
                                   np.linspace(y, first_point[1], 100)))
            self.Move_to_line_light.append(to_new_line)
            for points in line:
                x = points[0]
                y = points[1]


    def move_to_line_processing(self):

        Round_move_to_new_line_dark = round_points(self.Move_to_line_dark,
self.width_between_line_dark,self.width_y)
        Round_move_to_new_line_light = round_points(self.Move_to_line_light,
self.width_between_line_light, self.width_y)
        self.Final_move_to_new_line_dark =
remove_same_points_in_line(Round_move_to_new_line_dark,

self.get_round_005,
self.get_round_005,
self.width_between_line_dark,
self.width_y)
        self.Final_move_to_new_line_light =
remove_same_points_in_line(Round_move_to_new_line_light,
self.get_round_01,
self.get_round_01,
```

```
self.width_between_line_light,
self.width_y)

    def plot_points(self):
        fig = plt.figure(figsize=(7.5,8))
        plt.xlabel('X [cm]')
        plt.ylabel('Y [cm]')
        plt.xscale("linear")
        plt.yscale("linear")
        for line in self.Final_lines_dark:
            for point in line:
                plt.plot(point[0],point[1], marker='.', markersize=1,
color="black")
        for line in self.Final_lines_light:
            for point in line:
                plt.plot(point[0],point[1], marker = '.', markersize = 1, color =
'gray')
        # Draw manipulator arms
        # unutarnja ruka
        plt.plot([0, 0], [0, 10], linewidth=1, color='red')
        # vanjska ruka
        plt.plot([0, 10], [10, 10], linewidth=1, color='blue')
        plt.show()
        fig.savefig('Point_figure.png')

    def plot_drawing_lines(self):
        fig = plt.figure(figsize=(7.5,8))
        plt.xlabel('X [cm]')
        plt.ylabel('Y [cm] ')
        plt.xscale('linear')
        plt.yscale('linear')
        for line in self.Final_lines_dark:
            x_first = line[0][0]
            y_first = line[0][1]
            x_last = line[-1][0]
            y_last = line[-1][1]
            x = [x_first,x_last]
            y = [y_first,y_last]
            plt.plot(x,y,color= 'black', linewidth = 0.1)
        for line in self.Final_lines_light:
            x_first = line[0][0]
            y_first = line[0][1]
            x_last = line[-1][0]
            y_last = line[-1][1]
            x = [x_first, x_last]
            y = [y_first, y_last]
            plt.plot(x, y, color='black', linewidth = 0.1)
        # Draw manipulator arms
        # unutarnja ruka
        plt.plot([0, 0], [0, 10],linewidth = 1, color='red')
        # vanjska ruka
        plt.plot([0, 10], [10, 10], linewidth = 1, color='blue')
        plt.show()
        fig.savefig('Line_figure.png')

    def Save_results(self):

        self.name_of_txt_file = self.photo[:-4] + '_hatch.txt'
```

```
        with open(self.name_of_txt_file, 'w') as point_dic:
            points = {}
            points["dark_points"] = self.Final_lines_dark
            points["move_to_new_line_dark"] = self.Final_move_to_new_line_dark
            points["light_points"] = self.Final_lines_light
            points["move_to_new_line_light"] = self.Final_move_to_new_line_light
            json.dump(points, point_dic, indent=8)

    def Send_results(self):

        cmd = ['scp', 'C:\\Users\\Ivan\\PycharmProjects\\ploter\\'+
self.name_of_txt_file, 'pi@raspberrypi.local:']
        proc = subprocess.run(cmd, stderr=subprocess.PIPE, stdout=subprocess.PIPE,
shell=True)
```

Glavna *Python* skripta za obradu fotografije u kojoj je prikazan primjer iz rada kako obraditi fotografiju.

```
from Image_processing import *

image = 'mona.jpg'
mona_lisa = Image_processing(image)
mona_lisa.import_image()
mona_lisa.upper_frame()
mona_lisa.define_bounds()
mona_lisa.threshold_dark = 60
mona_lisa.threshold_light = 100
mona_lisa.getting_points_pixels()
mona_lisa.image_transformation()
mona_lisa.drawing_round_points()
mona_lisa.drawing_points_processing()
mona_lisa.get_move_to_new_drawing_line()
mona_lisa.move_to_line_processing()
mona_lisa.plot_points()
mona_lisa.plot_drawing_lines()
mona_lisa.Save_results()
mona_lisa.Send_results()

image = 'author.jpg'
author = Image_processing(image)
author.import_image()
author.upper_frame()
author.define_bounds()
author.threshold_dark = 90
author.threshold_light = 120
author.getting_points_pixels()
author.image_transformation()
author.drawing_round_points()
author.drawing_points_processing()
author.get_move_to_new_drawing_line()
author.move_to_line_processing()
author.plot_points()
author.plot_drawing_lines()
author.Save_results()
author.Send_results()
```

3. *Python* kod za vizualizaciju područja rada manipulatora bez matematičkog modela

```
from turtle import *
import math

inner_radius = 10
outer_radius = 10
extent1 = 360
extent2 = 360
joint_angle = 90
steps = 5
draw_arms_every = 1


class T(Turtle):

    def draw_inner_arm(self, angle):
        self.up()
        self.home()
        self.width(2)
        if (angle/draw_arms_every).is_integer() or angle==extent:
            self.down()
            self.color("blue")
            self.left(angle)
            self.fd(inner_radius * self.multiplier)
            self.dot(5, "black")
        else:
            self.left(angle)
            self.fd(inner_radius * self.multiplier)

    def draw_outer_arm(self):
        self.rt(joint_angle)
        self.color("red")

        self.fd(outer_radius * self.multiplier)
        self.fd(-outer_radius * self.multiplier)

    def draw_arc(self):

        self.up()
        self.rt(180)
        self.fd(outer_radius * self.multiplier)
        self.rt(-90)

        self.circle(outer_radius * self.multiplier, (360-extent1)/2)

        self.color("gray")
        self.down()
        self.width(3)
        self.circle(outer_radius * self.multiplier, extent1)


def visualise():

    s = Screen()
    s.setup(width=1000, height=1000, startx=0, starty=0)
```

```
        mode("logo")

        t = T()

        t.multiplier = 360/(inner_radius + outer_radius)

        t.speed(0)
        t.hideturtle()

        for angle in range (0, extent2+1, steps):
            t.draw_inner_arm(angle)
            t.draw_outer_arm()
            t.draw_arc()

        s.exitonclick()

    visualise()
```